Advanced User's Guide

Your Fast Track to Solutions

Microsoft, Windows, Windows NT, Windows 2000, Windows XP, Visual Basic, Microsoft .NET, Visual C++, Visual C#, and ActiveX are either trademarks or registered trademarks of Microsoft Corporation.

All other nationally and internationally recognized trademarks and tradenames are hereby recognized.

Copyright © 2002-2008 by MVTec Software GmbH, München, Germany

Edition 1	November 2002	(ActivVisionTools 2.1)
Edition 2	May 2003	(ActivVisionTools 2.2)
Edition 3	January 2005	(ActivVisionTools 3.0)
Edition 4	February 2006	(ActivVisionTools 3.1)
Edition 5	May 2008	(ActivVisionTools 3.2)

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

More information about ActivVisionTools can be found at:

`http://www.activ-vision-tools.com`

How to Read This Manual

This manual is addressed at the advanced user of ActivVisionTools. It focuses on topics that become relevant when developing complex ActivVisionTools applications, e.g., how to influence the execution sequence of the tools, how to create new ActivVisionTools via ActivGenericInterface, or how to customize the graphical user interface of your application using ActivGUIComponents. For all this functionality Visual Basic examples are provided.

To follow the examples actively, first install and configure ActivVisionTools as described in the manual Getting Started with ActivVisionTools. For each example in this manual, there is a corresponding Visual Basic project; these projects can be found in the subdirectory `examples\manuals\advanced_guide` of the ActivVisionTools base directory you selected during the installation (default: `C:\Program Files\MVTec\ActivVisionTools`). Of course, you can also create your own Visual Basic projects from scratch.

Furthermore, the manual describes how to use ActivVisionTools in other environments like Visual C++ and Visual Studio .NET. Example projects can be found in the subdirectories `examples\vcpp` and `examples\dotnet`.

We recommend to **create a private copy of the example projects** because by experimenting with the projects, you also change their *state*, which is then automatically stored in the so-called description files (extension `.dsc`) by ActivVisionTools. Of course, you can restore the state of a project by retrieving the corresponding description file from the CD.

Contents

1 Using ActivVisionTools in C++ and in .NET Languages **1**

1.1 Introduction 2

1.2 ActivVisionTools in Visual Studio .NET 3

1.3 ActivVisionTools in Visual C++ 14

2 Creating Customized GUIs **39**

2.1 Basics 40

2.2 Example 42

3 Influencing the Execution **53**

3.1 The Standard Execution Cycle 54

3.2 Triggering Execution Cycles 59

3.3 Modifying the Execution Cycle 63

3.4 Modifying the Input Data of Individual Tools 66

3.5 Handling DSC Files via the Programming Interface 83

3.6 Using Multiple Instances of ActivView 88

4 Creating Own ActivVisionTools **107**

4.1 Introducing ActivGenericInterface 108

4.2 Some Words About HALCON 108

4.3 How to Create a New ActivVisionTool 110

4.4 Examples 118

5 Advanced Image Acquisition **145**

5.1 Parallel Grabbing 146

5.2 Grabbing from Multiple Cameras 147

Chapter 1

Using ActivVisionTools in C++ and in .NET Languages

1.1 Introduction **2**

1.2 ActivVisionTools in Visual Studio .NET **3**

1.2.1 Performing Garbage Collection and Saving DSC Files 3

1.2.2 Using the Dialog Manager to Open Modeless Dialogs 4

1.2.3 Using ActivVisionTools With Other .NET languages 5

1.2.4 Additional Information 9

1.3 ActivVisionTools in Visual C++ **14**

1.3.1 Basic Steps 14

1.3.2 Creating an ActivVisionTools Application 16

1.3.3 Using the Dialog Manager to Open Modeless Dialogs 24

1.3.4 Accessing the Results 28

1.3.5 Using ActivVisionTools in Visual C++ .NET 36

1.3.6 Additional Information 37

1.1 Introduction

In the manual Getting Started with ActivVisionTools we remarked that, in theory, ActivVisionTools can be used in all environments that support ActiveX controls. These environments, however, differ significantly, thus it should come as no surprise that using ActivVisionTools in, e.g., Visual C++, is not as straightforward as in its "native" environment Visual Basic (version 6.0). The same applies to Visual Studio .NET, even to Visual Basic .NET.

Both Visual C++ and Visual Studio .NET impose two major restrictions:

- **No execution at design time**
  In contrast to Visual Basic 6.0, Visual C++ and Visual Studio .NET do not allow to edit ActiveX controls at design time. This means that you cannot execute an ActivVisionTools application at design time, because you cannot trigger actions like grabbing an image by clicking a button.

- **Only modal dialogs**
  In Visual C++ and Visual Studio .NET, all dialogs which are opened by ActiveX controls are modal, i.e., they block the application until closed. If you already worked with ActivVisionTools within Visual Basic 6.0, which does not impose this restriction, you know that this is a serious restriction, especially while configuring an application.

  To alleviate this problem, ActivVisionTools provides special support classes for Visual C++ and Visual Studio .NET, which allow you to open modeless dialogs interactively via a dialog manager. For Visual Studio .NET, these classes are also provided in a control that can be added to the form interactively, i.e., without programming.

As a further restriction, the mechanism for saving the state of an ActivVisionTools application, i.e., the current values of its parameters and properties and other information like ROIs etc., in the DSC file does not work as in Visual Basic 6.0:

- **DSC files are not saved automatically**
  In contrast to Visual Basic 6.0, where the application state is saved automatically, in Visual C++ you must add some lines of code. For Visual Studio .NET, this functionality is also contained in the control mentioned above.

- **DSC files are not saved at design time**
  In Visual C++ and Visual Studio .NET, DSC files can not be saved at design time, because of two reasons: First, you cannot add a corresponding line of code, because it would not be executed at design time. Secondly, in contrast to Visual Basic 6.0, ActivVisionTools currently does not provide Add-Ins for Visual C++ and Visual Studio .NET that determine the name and the start directory of the current project (and thus the name and path of the DSC file).

1.2 ActivVisionTools in Visual Studio .NET

This section contains detailed information about using ActivVisionTools in Visual Studio .NET, especially what to do if you don't want to use the control `AVTNETControl` described in the manual Getting Started with ActivVisionTools, section 3.1.2 on page 28. The procedures are explained for Visual Basic .NET; see section 1.2.3 on page 5 for information about other .NET languages. The referenced example projects can be found in the directory `examples\dotnet`.

Note that in the following, we will always refer to the class name of tools without the prefix Ax, which is added by the InterOp assembly. This is consistent with the name of the tool in the original ActivVisionTools library, and also with the name displayed as tooltip in the toolbox. However, the actual class name in .NET is, e.g., `AxAVTView` . All of these classes are derived from the class `AxHost`. You will see these names when you handle events or dynamically create instances of tools. On the other hand, we will use the prefix Ax when it is part of a specific instance name, e.g., `AxAVTView1`.

1.2.1 Performing Garbage Collection and Saving DSC Files

If you create an application without adding the control `AVTNETControl` and start and stop this application repeatedly, you will note that your settings have not be saved; the error box that pops up is related to this effect. Also, the grab speed becomes unreliable after some time because of the garbage collection. To get the functionality provided by the control, we must add some code to the form. To do so, right-click on `Form1` in the Solution Explorer and select `View Code`; a nearly empty code window appears (see figure 1.1).

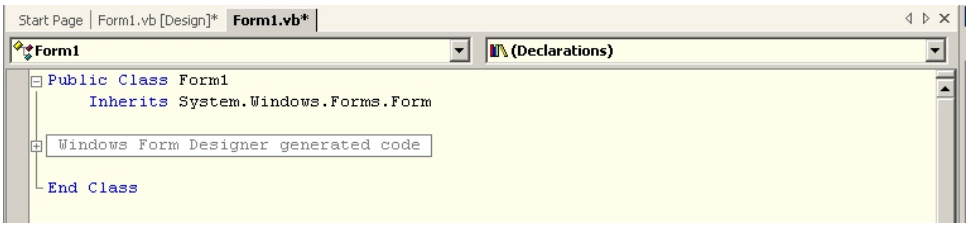


Figure 1.1: Initial state of the code of a form.

To perform the necessary garbage collection at the start of each execution cycle, you must add the following code into the class `Form1` after `Windows Form Designer Generated Code` (replace `AxAVTView1` by the actual instance name if necessary):

```
Private Sub AxAVTView1_CycleStart(ByVal sender As System.Object, _
                                  ByVal e As System.EventArgs) _
                                 Handles AxAVTView1.CycleStart
  Call GC.Collect()
  Call GC.WaitForPendingFinalizers()
End Sub
```

The fact that the settings are not automatically saved in the DSC file has a deeper-lying cause: Within Visual Studio .NET an ActivVisionTools application does not terminate automatically. Thus, you must add some more code to assure a clean termination upon closing the form:

```
Private Sub Form1_Closing(ByVal sender As Object, _
                          ByVal e As System.ComponentModel.CancelEventArgs) _
                         Handles MyBase.Closing
  Call AxAVTView1.ExitApplication(True)
End Sub
```

The method `ExitApplication` automatically saves the current state in the DSC file if you pass `'TRUE'` as a parameter. To test the effect of the code, run the application again. The warning box will popup for a last time. Select an image file and click the button `Single`. Now, when you close the form and restart the application, the changed settings will be used, which have been stored in a DSC file in the subdirectory `bin` of your project directory.

Alternatively, you can choose not to save upon termination by passing the value `'FALSE'` to `ExitApplication` in the event `Form_Closing`; besides this, you must set `AxAVTView` 's property `EnableWriteToDSCFile` to `'FALSE'` in the event `Form_Load`.

With `AxAVTView` 's method `WriteToDSC` you can save the application state at arbitrary moments; e.g., you could add a menu `File` with an item `Save`, which calls the method. Note however, that the method checks the property `EnableWriteToDSCFile`; thus, you must set the property to `'TRUE'` before calling the method.

For more details about handling DSC files please refer to section 3.5 on page 83.

1.2.2 Using the Dialog Manager to Open Modeless Dialogs

As already noted in section 1.1 on page 2, one of the main restrictions Visual Studio .NET imposes upon ActivVisionTools is that only modal dialogs can be opened by ActiveX controls. To test the effect of this restriction, run the application without the control `AVTNETControl` again. Right-click in the `AVTView` and open any dialog. Now, try to open a second dialog – it's not possible, because the first (modal) dialog blocks the application until it's closed.

You will find that it is inconvenient to use ActivVisionTools if you can only have one dialog open at a time. Thus, we have provided some support classes that allow you to open modeless dialogs. They are provided as Visual Basic .NET source code in the project `ActivVToolsNET`,

in case you need to customize them to your need. You can find this project in the directory `devenv\dotnet\vb\ActivVToolsNET` of the folder where you installed ActivVisionTools. It contains the following source files:

File	Description
AVTDialogManager.vb	Class for managing the opening of modeless ActivVisionTools dialogs
AVTDialog.vb	Windows Form intended to host an ActivVisionTools dialog
AVTDialogList.vb	Module containing information on available dialogs

You can integrate this functionality into your project by adding a reference to the assembly `ActivVToolsNET.dll` (see figure 1.2: Right-click `References` in the Solution Explorer and select `Add Reference`. Click on `Browse` and select the assembly `ActivVToolsNET.dll`, which is located in the subdirectory `devenv\dotnet\vb\ActivVToolsNET\bin` of your ActivVisionTools installation.

If you prefer, you can also add a reference to the `ActivVToolsNET` solution instead of the compiled assembly, or simply include the source files in your own project, but you need Visual Basic .NET to do so.

All you need to do get modeless dialogs is to install the dialog manager when your application is starting up.

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
                       Handles MyBase.Load
  Call ActivVToolsNET.AVTDialogManager.InstallDialogManager(Me)
End Sub
```

To test this, run your application, right-click on `AVTView` and select `Histogram`. ActivHistogram appears. Note that the dialog is modeless: You can right-click on `AVTView` to open further dialogs.

1.2.3 Using ActivVisionTools With Other .NET languages

So far, we concentrated on Visual Basic .NET. In principle, the other .NET languages should present no significant differences, because they all share the same .NET framework. Problems arise, however, if a language/development environment does not assist you with handling Windows Forms and ActiveX controls by providing designers and generating wrapper classes.

In addition to Visual Basic .NET, the current version of ActivVisionTools has been tested with Visual C# .NET and Visual C++ .NET.

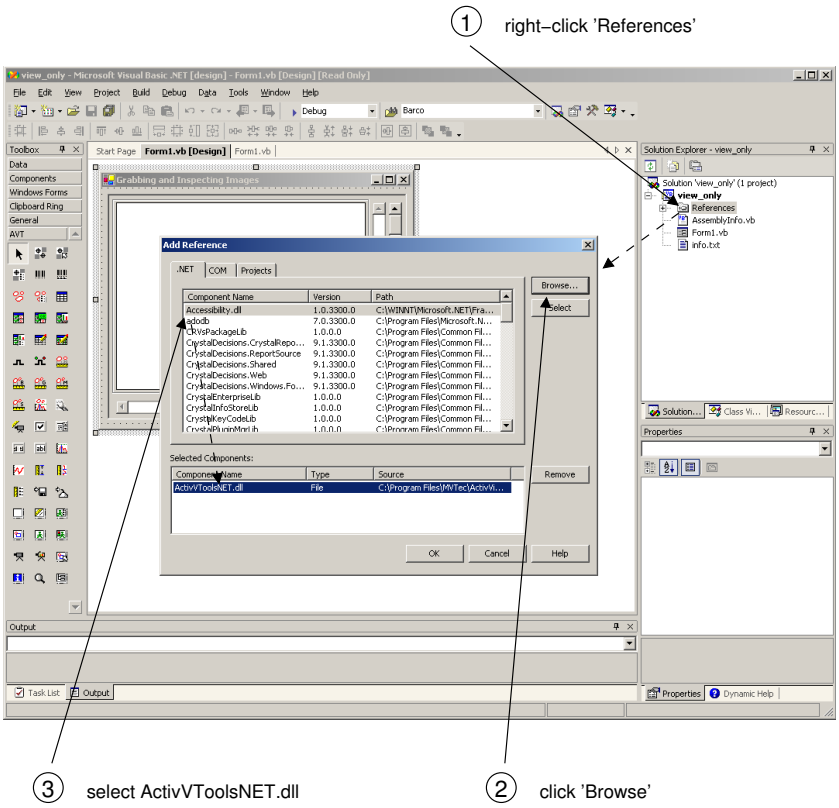


Figure 1.2: Adding `ActivVToolsNET.dll` .

1.2.3.1 Visual C# .NET

C# presents only minor syntactic differences. The corresponding handlers for the events `CycleStart`, `Form_Load`, and `Form_Closing` would look like this:

```
private void axAVTView1_CycleStart(object sender, System.EventArgs e)
{
  GC.Collect();
  GC.WaitForPendingFinalizers();
}
```

```
private void Form1_Load(object sender, System.EventArgs e)
{
  ActivVToolsNET.AVTDialogManager.InstallDialogManager(this);
}
```

```
private void Form1_Closing(object sender,
                           System.ComponentModel.CancelEventArgs e)
{
  axAVTView1.ExitApplication(true);
}
```

The following code registers these methods as event handlers. Note that this code is automatically added by the designer to the method `InitializeComponent`, if you insert an event handler by picking the desired event via the "lightning" icon in the property window.

```
private void InitializeComponent()
{
  ...
  this.axAVTView1.CycleStart +=
    new System.EventHandler(this.axAVTView1_CycleStart);
  ...
  this.Closing +=
    new System.ComponentModel.CancelEventHandler(this.Form1_Closing);
  this.Load += new System.EventHandler(this.Form1_Load);
  ...
}
```

Keep in mind that the DSC files, which store the settings, reside in the same directory as the executable. Therefore, you will need to **copy the DSC file manually when the output directory changes**. In particular, this can happen when the active configuration changes (Debug/Release).

The example project `vcs\view_only\view_only.cs` in the directory `examples\dotnet` is a simple ActivVisionTools application for image acquisition and inspection, which contains only ActivView.

1.2.3.2 Visual C++ .NET

Visual C++ .NET is different in that it has a strong focus on developing unmanaged code. For an unmanaged C++ application, you need to use the support functionality provided for Visual C++ 6.0 instead of that provided for .NET. See section 1.3 on page 14 for a detailed description; section 1.3.5 on page 36 explains the special steps necessary in Visual C++ .NET.

To create a managed Windows Forms application in Visual Studio .NET, you need Visual C++ 2005; we do not recommend to use ActivVisionTools with older versions of

Visual C++ .NET. Under Managed Visual C++ 2005, ActivVisionTools are used similarly as in C#. However, you cannot use the control `AVTNETControl` nor reference the DLL `ActivVToolsNET.dll`. The steps described in the following can be found in the example project `view_only_2005.vcproj` in the directory `examples\dotnet\vcpp\view_only_2005\view_only_2005`.

First, you must add the code for terminating the application and for garbage collection in the form's header file (in the example: `Form1.h`). The corresponding handlers for the events `CycleStart` and `Form_Closing` look like this:

```
private: System::Void axAVTView1_CycleStart(System::Object^  sender,
                                            System::EventArgs^  e)
       {
         GC::Collect();
         GC::WaitForPendingFinalizers();
       }
```

```
private: System::Void Form1_FormClosing(System::Object^  sender,
                             System::Windows::Forms::FormClosingEventArgs^  e)
       {
         axAVTView1->ExitApplication(true);
       }
```

To obtain modeless dialogs, add the class `AVTDialogs` to your project by right-clicking into the solution explorer, selecting `Add Existing Item ...`, navigating to `%AVTROOT%\devenv\dotnet\vcpp\include` and selecting the files `AVTDialogs.h` and `AVTDialogs.cpp`.

Then, apply the following additional modifications to the source code of your Form: Add the line

```
#include "AVTDialogs.h"
```

at the beginning of the file directly below `#pragma`. Furthermore, add an event handler for the event `OpenToolDialog` of `AxAVTView1` and insert code to read:

```
private: System::Void axAVTView1_OpenToolDialog(System::Object^  sender,
                             AxActivVTools::__AVTView_OpenToolDialogEvent^  e)
       {
         e->hWndDialog = AVTDialogs::OpenDialog(e->sToolClass, this);
       }
};
```

Finally, add `$(AVTROOT)\devenv\dotnet\vcpp\include\` to the include path which is used for locating header files. To do this, right-click on the project in the solution explorer and select `Properties`. You can find the property `include` under the node `C++`.

1.2.4 Additional Information

1.2.4.1 Restrictions

ActivVisionTools Property Changes in Design Mode are not Saved in the DSC File

This applies to changing ActivVisionTools properties via the property window in the designer of Visual Studio .NET. You should configure all properties of an ActivVisionTools application while the application is running.

You can conveniently change most settings in the normal GUI of the appropriate tool. For other settings, please use the tool `AVTViewProperties`, which provides a property window at run time. To configure the connections of tools and ActivGUIComponents, please use the tool `AVTViewConnections`. Both tools are available by right-clicking on `AVTView` and selecting `Connections` or `Connections`, respectively, in the appearing context menu (see the User's Manual for ActivView, section 2.3 on page 20 and section 3.5 on page 36 for more information about these tools).

Finally, a minor number of properties uses the standard persistence mechanism of ActiveX controls (which only works at design time). These properties are not stored in the DSC file, their values therefore only persist at design time:

- position and size of tools on the form
- `Appearance`, `BorderStyle` and `UseIconMode`
- `ForeColor` and `BackColor` (except for `AVTDecision` and `AVTDecisionResult` where the color carries run-time information)
- pictures used in GUI elements (e.g. by AVTGUICheckBox)
- all properties of ActivGenericInterface
- `AVTView`'s properties `UsePopUpAppl`, `UsePopUpConfig`, `EnableWriteToDSCFile`, and `UseZoomScrollBars`
- `AVTZoom`'s property `WindowSize`
- `AVTDecisionResult`'s properties `Style`, `BarOrientation`, `FullImageSize`, `UseDefaultImage`, and `UseBadImageOnError`. Note that you can conveniently access these properties using the associated property page (see the User's Manual for ActivDecision, section 2.5 on page 16). Settings which are not available at design time will be grayed out.

Some Dialogs are Only Available Modally

Section 1.2.2 on page 4 described how to use the `ActivVToolsNET` support classes to open modeless dialogs. However, this procedure only works for dialogs whose functionality is available as an ActiveX control. This is not the case for the following dialogs:

- history of `AVTViewStatus`
- configuration of `AVTDecisionResult`
- options and selection of output data for ActivFile and ActivSerial
- board settings for ActivDigitalIO
- input or output channel assignment for ActivDigitalIO

1.2.4.2 Further Examples

The directory `examples\dotnet` contains example projects for Visual Basic .NET, C#, and Visual C++ .NET, including code for the access of results via the programming interface of ActivVisionTools.

Please note that all examples in the directory `examples\dotnet` reference the control `AVTNET-Control`.

If you want to run the Visual Basic 6.0 examples in `examples\manuals` you must upgrade them to Visual Basic .NET as described below.

1.2.4.3 How to Upgrade Visual Basic 6.0 Projects

Most versions of Visual Basic .NET provide an upgrade wizard or migration tool for importing Visual Basic 6.0 projects. Note that this tool is far from perfect; in most cases you will be better off re-creating the application manually. Whether it works at all may depend on the version and on the complexity of the project.

In theory, you can upgrade an ActivVisionTools project from Visual Basic 6.0 to Visual Basic .NET using the upgrade wizard with the following steps:

1. In Visual Studio .NET, select `Open Project` and open the project file (file extension `.vbp`) of the Visual Basic 6.0 project. The Upgrade Wizard will execute.
2. The default behavior is to create the .NET project as a subdirectory of the original project directory. If this if ok, then accept all suggestions with `OK`, `Yes` or `Next`, until the upgrading starts.

3. During the upgrade, an ActivVisionTools warning will appear that complains about a lost DSC file. Click OK and ignore.
4. When upgrading is finished, the Upgrade Wizard will crash, opening two error boxes. Click OK and ignore, the job should be done.
5. Build the Solution. A dialog called `Save File` will pop up. Simply click `Save`, to create a solution file with the same name as the project file in the project directory.
6. Open a Windows Explorer and copy the DSC file from the old project directory into the subdirectory `bin` of the new project directory. Rename the DSC file so that the name matches the name of the executable.
7. If applicable, copy further data files for ActivAlignment (`.alm`), ActivDataCode (`.dmd`), or ActivOCR (`.ocr`) into the directory `bin` as well, but do not rename them.
8. Add `AVTNETControl` (see Getting Started with ActivVisionTools, section 3.1.2 on page 28) and run the application.

With the described procedure, we were able to upgrade most of our Visual Basic 6.0 example projects in `examples\manuals` in Visual Studio .NET 2002. Note that the following example projects are known to cause upgrade problems:

- **activmeasure\evalaccess, activbarcode\evalaccess, activdatacode\evalaccess, activblobfinder\evalaccess, activfeaturecalc\evalaccess, activdecision\access**
  The old and new classes `Collection` are confused, which prevents upgrading the event handler for the event `Finish` of ActivDecision.

  As a work-around, replace

  ```
  Private Sub AVTDecision1_Finish(atToolResults As Collection)
  ```

  with

  ```
  Private Sub AVTDecision1_Finish(ByVal eventSender As System.Object, _
                                  ByVal eventArgs As _
                                  AxActivVTools.__AVTDecision_FinishEvent) _
                                 Handles AVTDecision1.Finish
    Dim atToolResults As VBA.Collection = eventArgs.atToolResults
  ```

- **advanced_guide\contours**
  The behavior of check boxes changed, which leads to access of uninitialized components.

  As a workaraound, declare the variable

  ```
  Private mbIsInitializing As Boolean
  ```

In `Public Sub New()` in the region `Windows Form Designer generated code`, replace

```
InitializeComponent()
```

with

```
mbIsInitializing = True
InitializeComponent()
mbIsInitializing = False
```

In all four event handlers ending with `_CheckStateChanged`, enclose the code with

```
If Not mbIsInitializing Then
  ...
End if
```

- **activdigitalio\programming**
  The changed event order corrupts interprocess handshaking; there is no simple workaraound available.
- **activdigitalio\unsupported**
  The virtual demonstration board is a `UserControl` that can't be upgraded; there is no simple workaraound available.

1.2.4.4 Error Handling

Please note that .NET ignores errors that are raised in event handlers for ActiveX objects. If you write own code, e.g., to access results in the event `Finish`, or to implement custom image processing using ActivGenericInterface, it is important that you add error handling to the event handler by using a `try...catch` block or an `On Error` statement. Without this, you will not be informed about run-time errors in your code; instead, you will see follow-up errors in ActivVisionTools caused, e.g., by working on image data that remains uninitialized because of a bug in your handler.

1.2.4.5 Using ActivVisionTools Controls and Classes in Your Program

If you want to use ActivVisionTools controls (e.g., `AVTView`) or classes (e.g., `AVTToolResult`) in your program, we recommend to use the corresponding namespace, otherwise you must prefix the control and class names.

In Visual Basic .NET, you include the namespace for the controls with the following line:

```
Imports ActivVTools
```

For the classes in the type library, insert the line:

```
Imports ACTIVVTOOLSLib
```

In C#, this corresponds to the following lines:

```
using ActivVTools;
using ACTIVVTOOLSLib;
```

Note that the namespace `ActivVTools` also contains classes, e.g., `ActivVTools.AVTCalibClass`. You can check to which namespace a class belongs in the Reference Manual: Classes under `Tools` and `Classes` stem from `ActivVTools`, the others from `ACTIVVTOOLSLib`.

1.2.4.6 Using COM Interfaces

Sometimes, you might want to use a method belonging to the common tool interface `IAVTool-Class`. You can access the interface, e.g., of `AxAVTView1`, in Visual Basic .NET using

```
Imports ActivVTools
[...]
Dim iatTool As IAVToolClass = AxAVTView1.GetOcx()
```

In C#, use the following code:

```
using ActivVTools;
[...]
IAVToolClass iatTool = (IAVToolClass) axAVTView1.GetOcx();
```

You might wish to do this, e.g., if you need to assign an explicit predecessor using the method `ConnectTo` (see also section 1.2.4.1 on page 9).

1.3 ActivVisionTools in Visual C++

Using ActivVisionTools in C++ is not as intuitive as in Visual Basic. This section explains the required steps. It concentrates on Visual C++ 6.0; section 1.3.5 on page 36 describes additional steps necessary when using Visual C++ .NET. The referenced example projects can be found in the directory `examples\vcpp`.

1.3.1 Basic Steps

First, create a Visual C++ project as follows:

- Open Visual C++.
- Select the menu item `File` ▷ `New`. In the tab `Projects`, select `MFCAppWizard` from the list. Choose a name for the project and press `OK`.
- The MFC AppWizard guides you through the following steps:
  1. In step 1, select the application type for your project (see figure 1.3). Typically, you select `Dialog based`. If you want to include a toolbar or menu, you should select `Single document`.
  2. In step 2, no changes are necessary.
  3. In step 3, no changes are necessary.

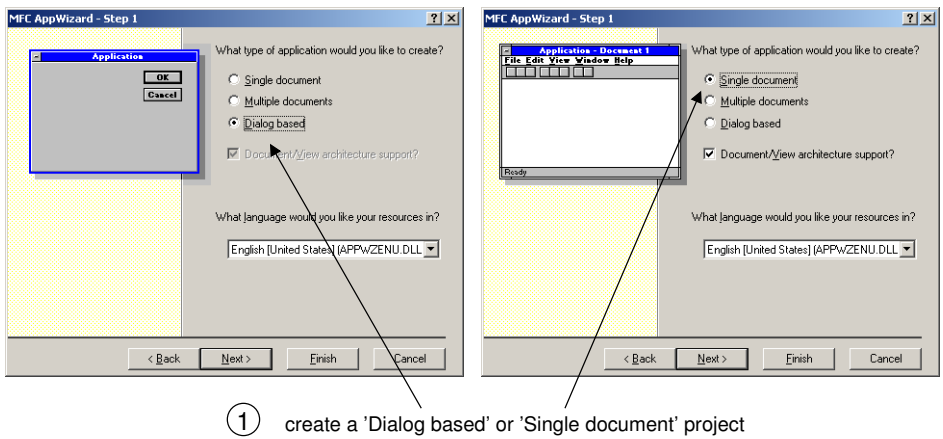


Figure 1.3: Creating a project.

4. In step 4, no changes are necessary if you selected `Dialog based` in step 1. If you selected `Single document`, press `Advanced` if you want that your application can't be resized. Select the tab `WindowStyles` and uncheck `Thick frame`, then close it.

5. In step 5, no changes are necessary.

6. In step 6, no changes are necessary if you selected `Dialog based` in step 1. If you selected `Single document`, select `'CFormView'` in the combo box `Base Class`.

Finally, press `Finish`.

Please note that the default behavior of a standard dialog is to close when `Enter` or `ESC` is pressed. This means that a dialog-based application will terminate if you press these keys while the main dialog has the focus. The dialog-based example projects also show this effect.

You can override this behavior by including the following two lines in the header file of the dialog <*Project Name*>`Dlg.h` in a public section:

```
class CBarcode_evalaccessDlg : public CDialog
{
public:
  ...
  void OnOK() {}
  void OnCancel() {}
  ...
};
```

However, this effectively prevents the dialog from closing at all, so you also must to explicitly insert a handler for the message `WM_CLOSE` for the dialog (figure 1.6 on page 20 shows how to do this for another message, `WM_DESTROY`). Then, edit the code and change the call `CDialog::OnClose` to `CDialog::OnOK` to achieve the proper termination of the dialog:

```
void CBarcode_evalaccessDlg::OnClose()
{
  // TODO: Add your message handler code here and/or call default

  CDialog::OnOK();
}
```

1.3.2 Creating an ActivVisionTools Application

After creating a suitably configured Visual C++ project, we now show how to turn it into an ActivVisionTools application.

1.3.2.1 Adding `AVTView`

The first step towards an ActivVisionTools application is to add the main tool, `AVTView`, to the form (see figure 1.4):

1. Open the form (which is called dialog here), which represents your main application window, e.g., by clicking with the right mouse button on its entry in the browser window and selecting `Open`.
2. Right-click on the dialog and select `Insert ActiveX Control`.
3. Now select the desired ActivVisionTool, i.e., `AVTView`, and press `OK`.

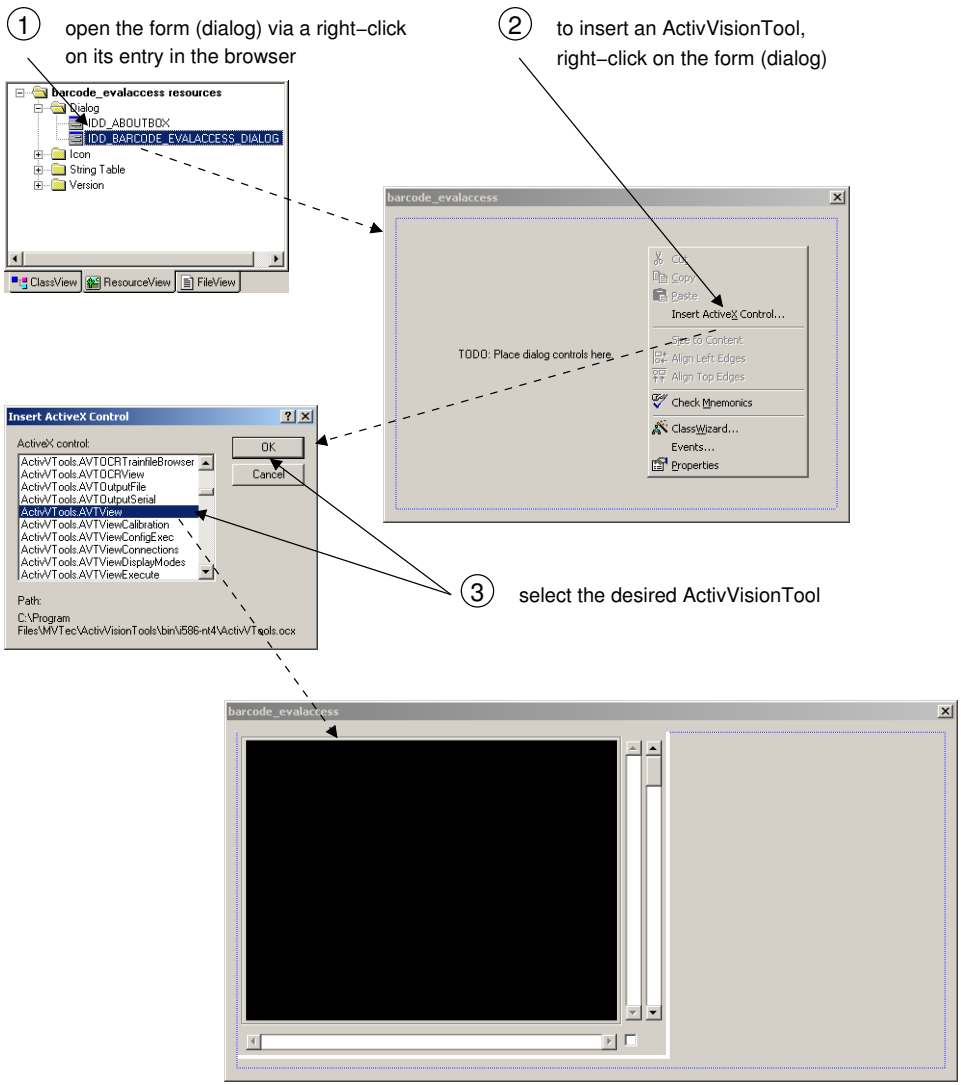


Figure 1.4: Adding AVTView.

1.3.2.2 Adding a Member Variable for `AVTView`

As described in section 1.3.2.3 on page 20, we must add some code to work around some restrictions of Visual C++. For this, we need a member variable for `AVTView`. First, open the ClassWizard via the menu item `View` ▷ `ClassWizard`. The following steps are visualized in figure 1.5.

1. Select the tab `Member Variables`.
2. If necessary, select the class `C`<*`Project Name`*>`Dlg` (for `Dialog based` applications) or `C`<*`Project Name`*>`View` (for `Single document` applications).
3. If necessary, select `IDC_AVTVIEW1` in the listbox `Control IDs` and press `Add Variable`.
4. In the first appearing dialog, press `OK`. In the second, uncheck all items except `C_AVTView` and press `OK`. This generates a wrapper class for `AVTView`.

   The other classes are offered because `AVTView` has properties or method arguments of the corresponding types. You will not need these wrapper classes unless you use such a method or property. If no wrapper classes are created, the corresponding parameters are treated as `IDispatch` interfaces. Note that the wrapper class generated for `AVTToolRegisterClass` will cause a compile error; this can be fixed by removing the offending method from the wrapper class.
5. In the next dialog, assign a name to the variable and press `OK`. Then, exit the ClassWizard by pressing `OK`.

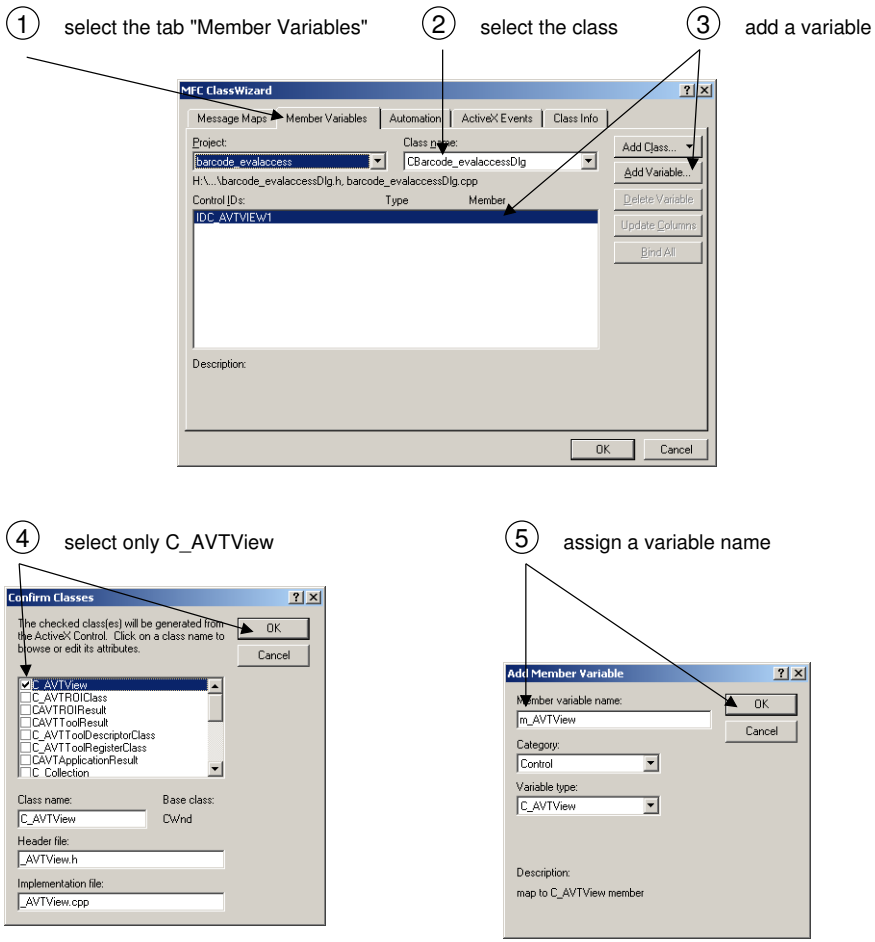


Figure 1.5: Adding a member variable for AVTView.

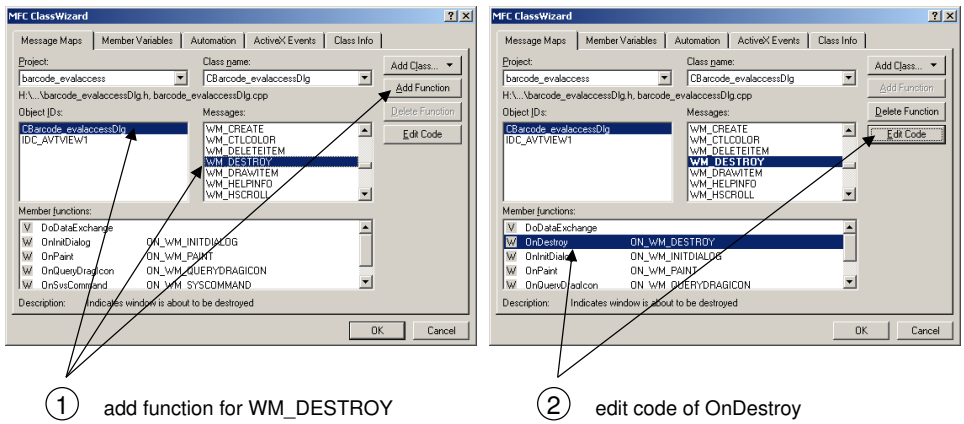


Figure 1.6: Adding a handler for the event `OnDestroy`.

1.3.2.3 Adding Necessary Code

As already noted, within Visual C++ an ActivVisionTools application does not run as out-of-the-box as within Visual Basic 6.0. Thus, you must add some code manually.

Correct Termination

Within Visual C++, an ActivVisionTools application does not exit cleanly. To add the necessary code to a `Dialog based` application, open the ClassWizard again. In the tab `Message Maps`, select the class C<*Project Name*>Dlg (for `Dialog based` applications) or C<*Project Name*>View (for `Single document` applications) in the listbox `Objects IDs`. From the entries in `Messages` select `WM_DESTROY`, press the `Add Function` and then `Edit Code` (see figure 1.6). Modify the generated code to obtain the following:

```
void CBarcode_evalaccessDlg::OnDestroy()
{
  if (m_AVTView)
    m_AVTView.ExitApplication(TRUE);

  CDialog::OnDestroy();
}
```

For a `Single document` application, the code looks like this:

```
void CAVTSingleDocumentView::OnDestroy()
{
  if (m_AVTView)
    m_AVTView.ExitApplication(FALSE);

  CFormView::OnDestroy();
}
```

Pass `'TRUE'`to the method `ExitApplication` if you want that the application automatically saves the current state in the DSC file upon exiting.

Missing DSC Files

When you now start the application for the first time, a message box similar to the one depicted in figure 1.7 pops up, telling you that no DSC file has been found. Upon exiting the aplication, the DSC file will be created, so that the message box does not appear again. Keep in mind that the DSC file that stores the settings resides in the same directory as the executable. Therefore, you will need to **copy the DSC file manually when the output directory changes**. In particular, this can happen if the active configuration changes (Debug/Release).

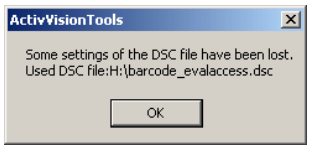


Figure 1.7: Dialog signaling a missing DSC file.

Disable Automatic Saving

You can disable the automatic saving of the application's state in the DSC file by inserting the following code in the method `OnInitDialog` (`Dialog based` application) or `OnInitialUpdate` (`Single document` application), respectively:

```
if (m_AVTView)
  m_AVTView.SetEnableWriteToDSCFile(FALSE);
```

Save State Upon Manual Saving (Only `Single document`)

Within Visual Basic 6.0, the state of an application is automatically saved in the DSC file not only upon termination, but also whenever you save the application itself. If you inserted a toolbar or menu in your (`Single document`) application that provides an icon or menu entry for saving the application, you can attach the code for saving the DSC file as follows: Open the ClassWizard again. Select the settings depicted in figure 1.8 and then press `Add Function`. This will insert the handler `OnFileSave`. Now press `Edit Code` and insert the following code:

```
void CAVTSingleDocumentView::OnFileSave()
{
  BOOL bOldValue;

  if (m_AVTView)
  {
    bOldValue = m_AVTView.GetEnableWriteToDSCFile();
    if (!bOldValue)
      m_AVTView.SetEnableWriteToDSCFile(TRUE);
    m_AVTView.WriteToDSC();
    m_AVTView.SetEnableWriteToDSCFile(bOldValue);
  }
}
```

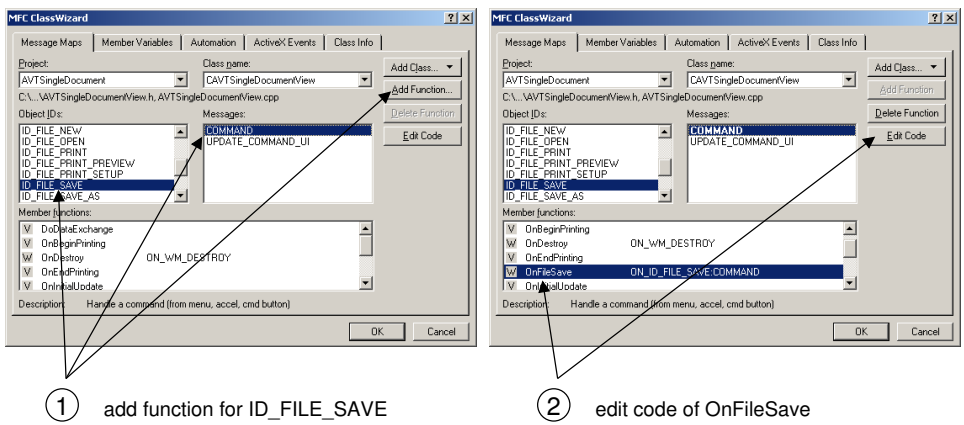


Figure 1.8: Adding a handler for the event `OnFileSave`.

Remove Unused Entries from Own Menu and Toolbar (Only `Single document`)

If you inserted a toolbar or menu in your (`Single document`) application, we recommend to suppress the menu items `File` ▷ `New` and `File` ▷ `Open` and the corresponding toolbar icons because they are not used by an ActivVisionTools application. Furthermore, we neither need the popup menu item `Edit`, nor the entries for printing. You can remove entries from menu or toolbar in the `ResourceView` in the tree item `Menu` or `Toolbar`, respectively. Figure 1.9 depicts this dialog and the modified menu and toolbar.

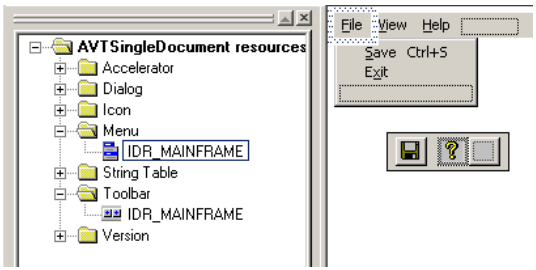


Figure 1.9: Menu and toolbar after removing unused entries.

1.3.2.4 Changing the Title of the Application (Only `Single document`)

In `Dialog based` applications, you can change the title interactively via the properties of the dialog. For `Single document` applications, in contrast, you must add some code to the method `InitInstance` in the file <*Project Name*>`.cpp`. Near the end of the method, insert a call to the method `SetWindowText` as in the following code fragment:

```
BOOL CAVTSingleDocumentApp::InitInstance()
{
  ...

  // change the title
  m_pMainWnd->SetWindowText("bar code");

  // The one and only window has been initialized, so show and update it.
  m_pMainWnd->ShowWindow(SW_SHOW);
  m_pMainWnd->UpdateWindow();

  return TRUE;
}
```

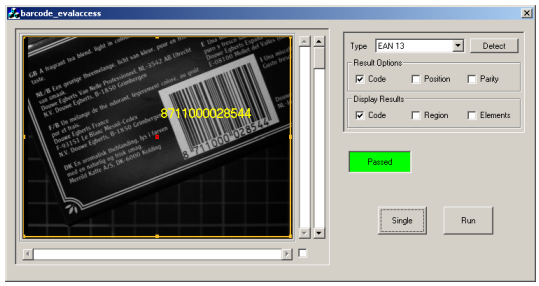


Figure 1.10: An application inspecting bar codes.

1.3.2.5 Adding Other ActivVisionTools

You add other ActivVisionTools to your application as described for `AVTView` in section 1.3.2.1 on page 16. Note that in contrast to `AVTView`, you don't need to create member variables and wrapper classes for all tools. Initially, all you need to do is to insert the ActiveX controls. Only if you need to access a tool via programming, e.g., to access the results that have been calculated, you must create a member variable and wrapper classes; this is described in section 1.3.4 on page 28.

The example bar code application in figure 1.10 can be created in a few simple steps: First, add `AVTBarcode1D`, `AVTDecision`, and two instances of `AVTViewExecuteSimple` and start the application. Open `AVTViewProperties` via the context menu of `AVTView` and configure one instance of `AVTViewExecuteSimple` as a single step button (see User's Manual for ActivView, section 3.4 on page 34). Close `AVTViewProperties` and execute a single step to load the first image of `barcode\barcode1.seq`. Now, click `Detect` in `AVTBarcode1D`; after creating an ROI over the whole image, the bar code is detected and read (please consult the User's Manual for ActivBarcode for more information about reading bar codes).

To evaluate the bar codes, open the dialog `AVTDecisionViewResults` via the context menu of `AVTDecision`. In this dialog, check the box `√` `Enable Update` and enter the conditions depicted in figure 1.11 (please consult the User's Manual for ActivDecision for more information about this tool). Then, close `AVTDecisionViewResults`. If you now step through the image sequence, you can see that some images contain "incorrect" bar codes.

1.3.3 Using the Dialog Manager to Open Modeless Dialogs

As already noted in section 1.2 on page 3, one of the main restrictions Visual C++ imposes upon ActivVisionTools is that only modal dialogs can be opened by ActiveX controls. To test the effect of this restriction, run your application, right-click in the `AVTView` and open any dialog. Now, try to open a second dialog – it's not possible, because the first (modal) dialog blocks the

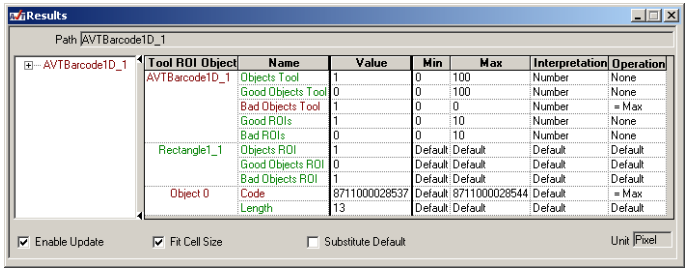


Figure 1.11: Conditions for inspecting the bar codes.

application until it's closed.

You will find that it is inconvenient to use ActivVisionTools if you can only have one dialog open at a time. Thus, we have provided some support classes that allow you to open modeless dialogs. The corresponding files are located in the directory `devenv\vcpp` of the folder where you installed ActivVisionTools:

1. **dialog templates and classes**
   in the subdirectory `AVTExt\bin`:
   - `AVTExt.dll`
   - `AVTExtD.dll` (debug version)
   - `AVTExtU.dll` (Unicode version)
   - `AVTExtUD.dll` (Unicode debug version)
   - `AVTExt70.dll`, `AVTExt70D.dll`, `AVTExt70U.dll`, `AVTExt70UD.dll`, `AVTExt71.dll`, `AVTExt71D.dll`, `AVTExt71U.dll`, `AVTExt71UD.dll`, `AVTExt80.dll`, `AVTExt80D.dll`, `AVTExt80U.dll`, `AVTExt80UD.dll` (DLLs for Visual C++ .NET 7.0, 7.1, and 8.0, respectively)

   in the subdirectory `AVTExt\lib`:
   - `AVTExt.lib`
   - `AVTExtD.lib` (debug version)
   - `AVTExtU.lib` (Unicode version)
   - `AVTExtUD.lib` (Unicode debug version)
   - `AVTExt70.lib`, `AVTExt70D.lib`, `AVTExt70U.lib`, `AVTExt70UD.lib`, `AVTExt71.lib`, `AVTExt71D.lib`, `AVTExt71U.lib`, `AVTExt71UD.lib`, `AVTExt80.lib`, `AVTExt80D.lib`, `AVTExt80U.lib`, `AVTExt80UD.lib` (libraries for Visual C++ .NET 7.0, 7.1, and 8.0, respectively)

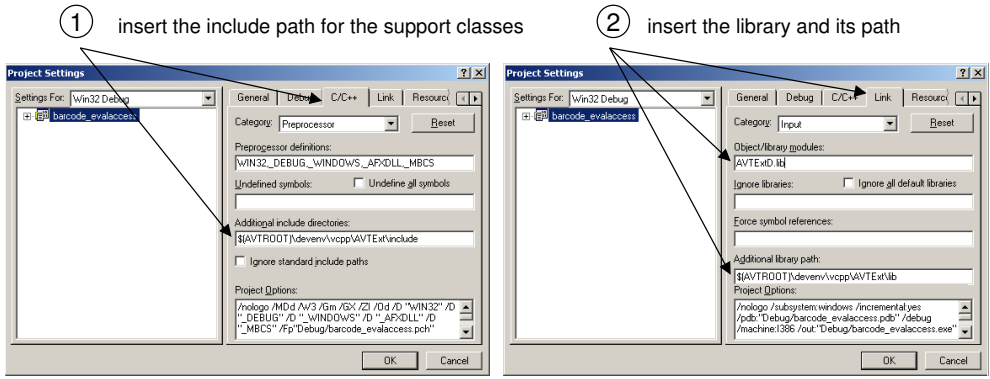


Figure 1.12: Specifying information for the compiler.

Please note that Visual Studio 2005, in contrast to previous versions, sets the character set for MFC applications by default to Unicode! Thus, you must link the Unicode versions of the libraries, e.g., `AVTExt80UD.lib` for a debug version. Alternatively, you can set the character set to `Use Multi-Byte Character Set` by opening the Solution Explorer by clicking on the solution with the right mouse button and selecting `Properties` and then selecting the general configuration properties.

in the subdirectory `AVTExt\include`:

- `AVTExt.h`

2. **Visual C++ .NET projects** (subdirectory `AVTExt`)
   - `AVTExt70.vcproj`, `AVTExt71.vcproj`, `AVTExt80.vcproj`

In the following sections, we show how to use these extensions in your ActivVisionTools application.

1.3.3.1 Adaptations for the Compiler

The first step is to tell the compiler where the required files are located. For this, open the menu entry `Project` ▷ `Settings` and perform the following steps (see figure 1.12):

① Select the tab `C/C++`. In the combo box `Category`, select `'Preprocessor'`. Under `Additional include directories`, insert the following path:

```
$(AVTROOT)\devenv\vcpp\AVTExt\include
```

We recommend to use the environment variable AVTROOT, which is set to the folder where you installed ActivVisionTools; note that, when using this variable in the compiler settings, it must be specified in the notation $(AVTROOT) as shown above.

② Then, select the tab Link. In the combo box Category, select 'Input'. Insert the path $(AVTROOT)\devenv\vcpp\AVTExt\lib under Additional library path. Under Object/library modules, insert the appropriate library from table 2. For example, for a release version insert AVTExt.lib.

Build Target	Corresponding Library
Release Version	AVTExt.lib
Debug Version	AVTExtD.lib
Release Unicode Version	AVTExtU.lib
Debug Unicode Version	AVTExtUD.lib

Table 1.1: Which library to use for which build target.

1.3.3.2 Include the Dialog Manager

In your application, you must include the header file for the dialog manager by inserting the following line in the file <*Project Name*>Dlg.cpp (for Dialog based applications) or <*Project Name*>View.cpp (for Single document applications):

```
#include "AVTExt.h"
```

1.3.3.3 Using Modeless Dialogs

To open modeless dialogs, you must insert an event handler for the event OpenToolDialog thrown by AVTView. For this, open the ClassWizard and select IDC_AVTVIEW1 in the list box Object IDs. Then, double-click on the entry OpenToolDialog in the list box Messages and press OK in the appearing dialog to add the event handler. Figure 1.13 depicts the result of this action. Finally, press Edit Code to edit the event handler. Insert the following code:

```
void CBarcode_evalaccessDlg::OnOpenToolDialogAvtview1(LPCTSTR sToolClass,
                              long FAR* hWndDialog)
{
  AVTDlgCreateAndShow(sToolClass, hWndDialog);
}
```

If you configure the bar code application described in section 1.3.2.5 on page 24 in this way,

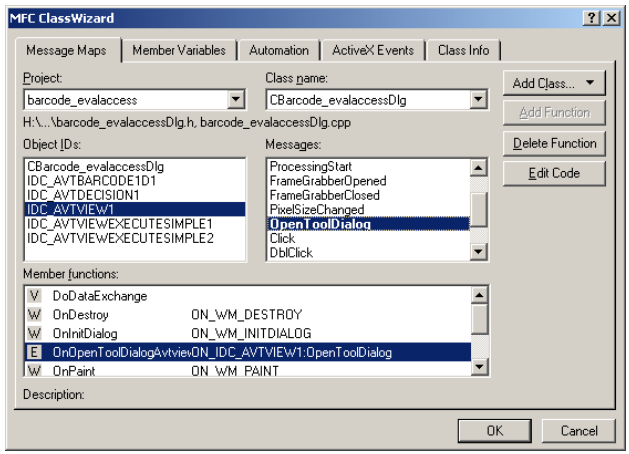


Figure 1.13: Inserting an event handler for `OpenToolDialog`.

you do not need to close the dialogs before opening other ones.

1.3.4 Accessing the Results

In this section we show how to access the results of an ActivVisionTool via the programming interface. As an example task, we extend the bar code application described in the previous section with a textbox that is to display the extracted code.

1.3.4.1 Adding a Member Variable and Wrapper Classes for `AVTBarcode1D`

To access the results of ActivBarcode, you must first create a member variable and wrapper classes for `AVTBarcode1D`. For this, open the ClassWizard and select the tab `Member Variables`. The following steps are visualized in figure 1.14.

(1) In the listbox `Control IDs`, double-click `IDC_AVTBARCODE1D1`.

! (2) In the first appearing dialog, press `OK`. In the second, **uncheck the item** `C_AVTView`. As noted beside the listbox when you select this line, `C_AVTView` has already been generated; you must not overwrite it, because the created wrapper class would have the wrong base class (`COleDispatchDriver`, as you can see by comparing the depicted dialog with the one depicted in figure 1.5 on page 19 (base class `CWnd`).

Close the dialog by pressing `OK`.

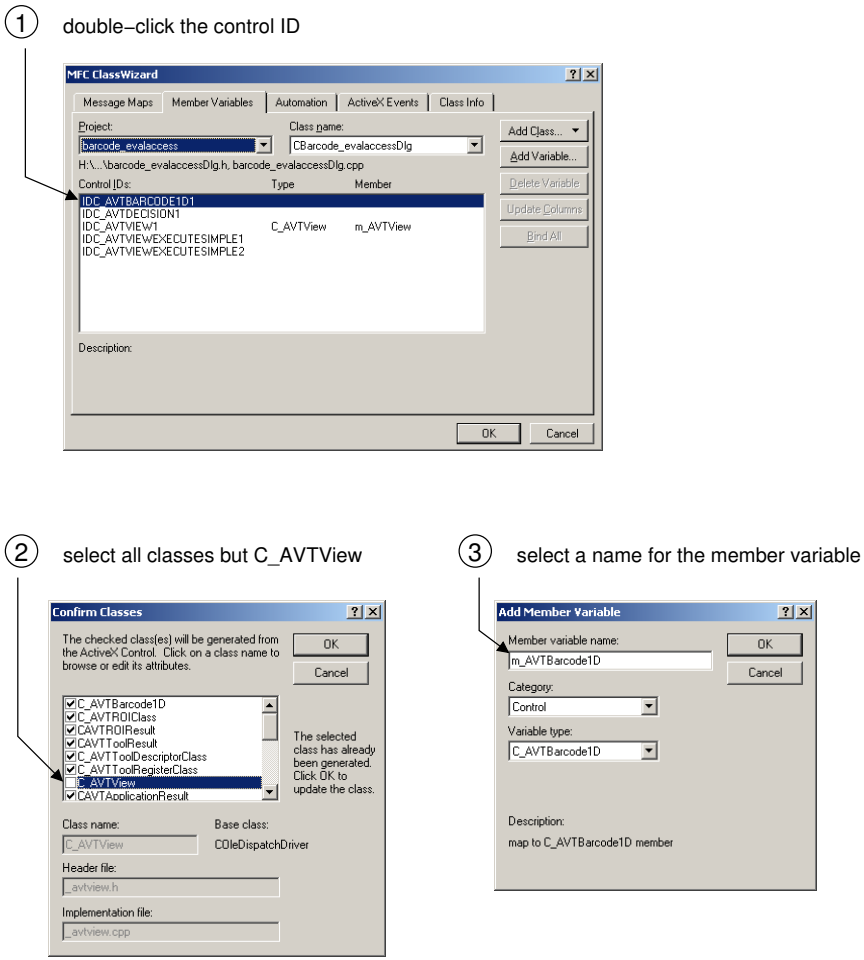


Figure 1.14: Adding a member variable and wrapper classes for `AVTBarcode1D` .

③ In the next dialog, assign a name to the variable and press OK, then close the ClassWizard by clicking OK here as well.

1.3.4.2 Adding a Textbox to Display the Extracted Code

Before we show how to access the main result of ActivBarcode, i.e., the extracted code, we add a textbox to the application to display the accessed value. The following steps are visualized in figure 1.15.

1. Add a textbox control to the application.
2. To open the property page of the new textbox, right-click it and select `Properties`.
3. Set its `ID` to `IDC_STATIC_CODE` and close the property page.
4. Now, open the ClassWizard and select the tab `Member Variables`. In the listbox `Control IDs`, select `IDC_STATIC_CODE` and then press `Add Variable`.
5. In the next dialog, assign a name to the variable and press `OK`. Then, the ClassWizard should look as depicted in the figure. Close it by clicking `OK`.

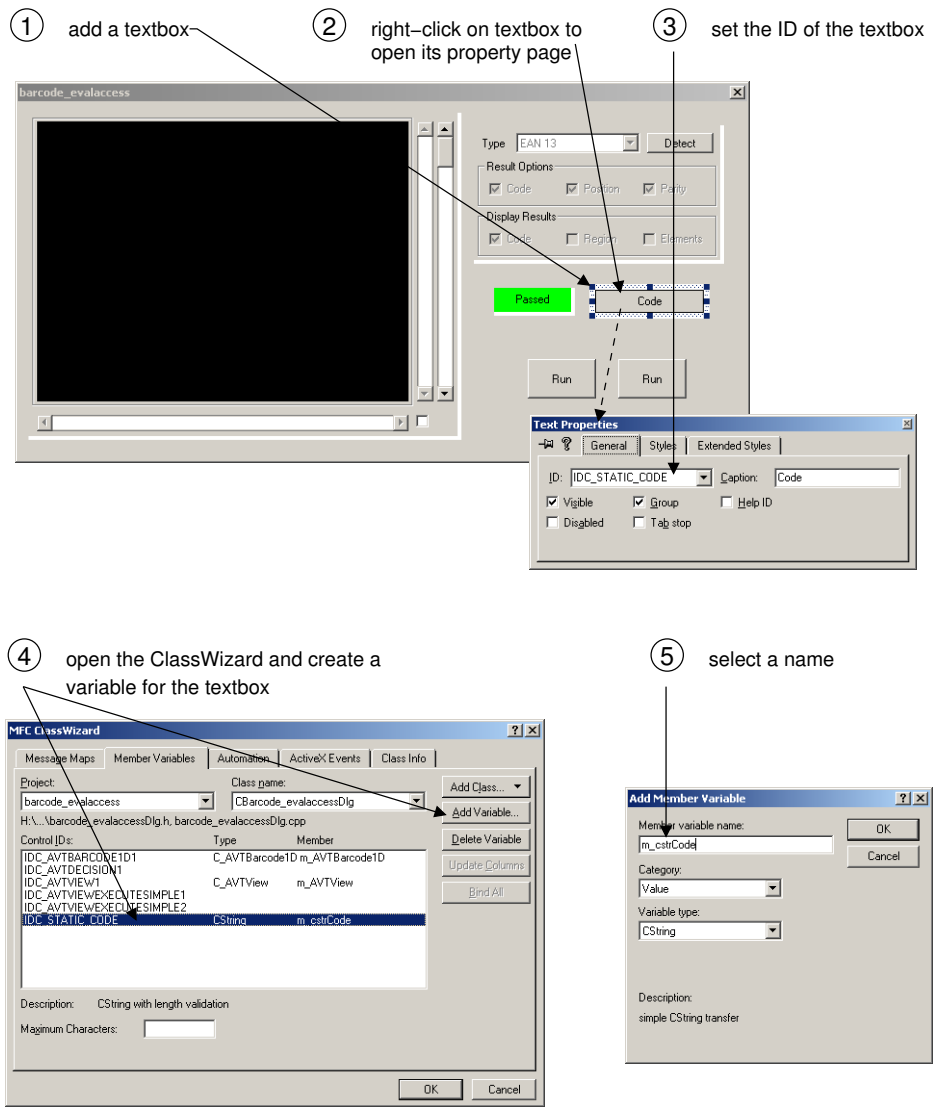


Figure 1.15: Adding a textbox and a corresponding variable.

1.3.4.3 Accessing an Extracted Bar Code

As described in the User's Manual for ActivBarcode, section 4.3.1 on page 27, the results of `AVTBarcode1D` can be accessed by creating a handler for its event `Finish`. To create this event handler in Visual C++, open the ClassWizard, switch to the tab `Message Maps`, double-click the event and click `OK` in the appearing dialog. Figure 1.16 shos the result.

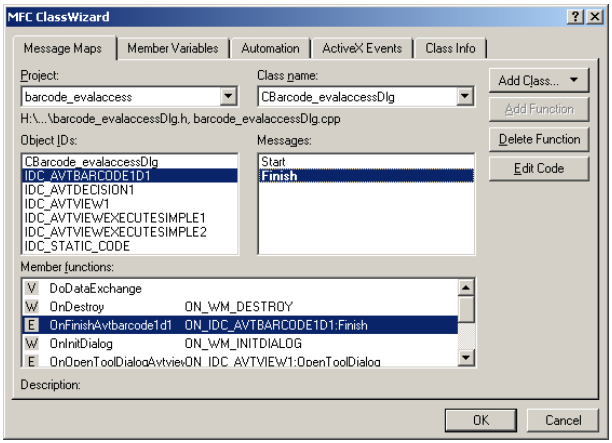


Figure 1.16: Adding a handler for `AVTBarcode1D`'s event `Finish` .

Automatically, an empty event handler is added to the file <*Project Name*>`Dlg.cpp` (for `Dialog based` applications) or <*Project Name*>`View.cpp` (for `Single document` applications), which you can open by pressing `Edit Code`.

The following code traverses the result tree of `AVTBarcode1D` to access the code of the first object of the first ROI and then displays the code in the textbox. We do not explain it in detail; please refer to the User's Manual for ActivBarcode, section 4.3.1 on page 27, for the basic mechanism of accessing results and compare the corresponding Visual Basic code examples. Similar code can be found in the example project `measure_evalaccess\measure_evalaccess.dsp`, which contains all the extensions described above.

Note that to use the ActivVisionTools classes you must first include the corresponding header files:

```
#include "AVTToolResult.h"
#include "AVTROIResult.h"
```

```
void CBarcode_evalaccessDlg::OnFinishAvtbarcode1d1(LPDISPATCH FAR*
                                                    atToolResults)
{
  try {
    CAVTToolResult avtToolResult(*atToolResults); // Wrapper does no AddRef!
    avtToolResult.m_bAutoRelease = FALSE; // => no Release, because no AddRef!

    int iNumROIs = avtToolResult.GetROINum();
    if (iNumROIs) {
      _variant_t_ext vROIResults;
      _variant_t_ext vROIResult;
      _variant_t_ext vValue;

      // get the ROIResults
      vROIResults.Attach(avtToolResult.GetROIResults());

      // access ROI 0
      vROIResult.Attach(vROIResults.GetElement(0));

      // determine number of objects for the ROI
      CAVTROIResult avtROIResult(vROIResult); // Wrapper does no AddRef!!
      avtToolResult.m_bAutoRelease = FALSE;

      int iNumObjects = avtROIResult.GetObjectNum();
      if (iNumObjects) {
        // get the value for the feature 'code'
        short sFeatIndex = m_AVTBarcode1D.FeatureHandleCode();
        vValue.Attach(avtROIResult.GetObjectValue(sFeatIndex, 0));
        m_cstrCode = CString( (TCHAR*)_bstr_t(vValue));

        // display the found code
        SetDlgItemText(IDC_STATIC_CODE,m_cstrCode);
        } else {
        m_cstrCode = CString("---");
        SetDlgItemText(IDC_STATIC_CODE,m_cstrCode);
      }
    }
  } catch (_com_error &e) {
    TRACE("%s\n", e.Description());
  }
}
```

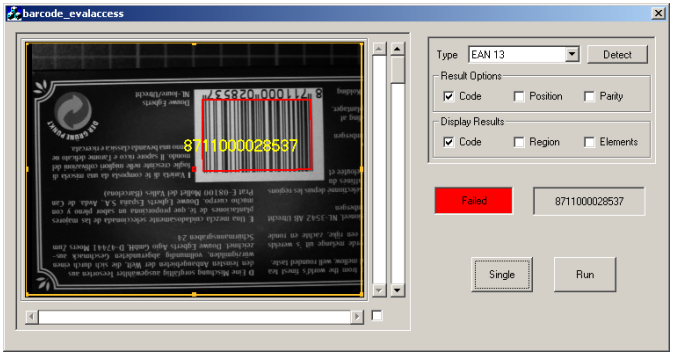


Figure 1.17: Final version of the bar code application.

Figure 1.17 shows the running application, with the extracted code being displayed in the textbox.

1.3.4.4 Accessing the Results of ActivDecision

The example project `measure_evalaccess\measure_evalaccess.dsp` includes code that accesses the evaluations via the programming interface. In fact, this example is the C++ version of the example used in the User's Manual for ActivMeasure, section 4.3.1 on page 31, which explains how to access results and evaluations via Visual Basic 6.0. We recommend to read the corresponding section and to compare the code fragments.

As in Visual Basic, you can access the results of ActivDecision, i.e., the evaluations, within a handler to `AVTDecision`'s event `Finish`. Note that as parameter you get an `LPDISPATCH*` that points to a `Collection` of `AVTToolResult` objects, one for each ActivVisionTool that generates feature results. The following code, taken from the file `measure_evalaccess\measure_evalaccessDlg.cpp` in the example's directory, shows how to extract the results of ActivMeasure:

As with the access of results, you must first include some header files to use the ActivVisionTools classes:

```
#include "AVTToolResult.h"
#include "AVTROIResult.h"

#include "_collection.h"
```

```
void CMeasure_evalaccessDlg::OnFinishAvtdecision1(LPDISPATCH FAR* atToolResults)
{
  try {
    CAVTToolResult avtToolResult;
    ...
    C_Collection atcolTRs;
    atcolTRs.AttachDispatch(*atToolResults, FALSE);

    lNumElements = atcolTRs.Count();
    if (!lNumElements)
      return;
    bResultsFound = true;

    _variant_t vtIndex;
    _variant_t vtTR;

    bResultsFound = true;
    vTRs = _variant_t_ext(*atToolResults);
    for ( i = 1 ; i <= lNumElements; i++) {
      vtIndex = _variant_t(long(i));
      // get the AVTToolResult
      vtTR = atcolTRs.Item(&vtIndex);
      avtToolResult.AttachDispatch(vtTR,FALSE);

      cstrName = avtToolResult.GetName();
      if (cstrName == "AVTMeasure1") {
        bResultsFound = true;
        break;
      }
    }

    ... access results and evaluations

}
```

1.3.5 Using ActivVisionTools in Visual C++ .NET

As already noted in section 1.2.3 on page 5, to create an ActivVisionTools application in Visual C++ .NET, we recommend to use the C++ support classes instead of the ones provided for .NET. Because Visual C++ 6.0 and Visual C++ .NET are based upon different MFC versions, separate DLLs and libraries are provided for the use in the latter, called `AVTExt70.dll`, `AVTExt70.lib` etc. (see the list in section 1.3.3 on page 24). The DLLs are provided for the versions 7.0, 7.1, and 8.0 (also called Visual C++ .NET 2002, Visual C++ .NET 2003, and Visual C++ 2005, respectively).

If you upgrade to a new version of MFC, you must generate new libraries by recompiling the project `AVTExt70.sln`, which is located in the subdirectory `AVTExt`.

Note that the exact procedure for inserting ActiveX controls and generating wrapper classes for them in Visual C++ .NET is somewhat different. Of the three different ways to generate wrapper classes in Visual C++ .NET, we recommend to use the following procedure:

- When you need to create a member variable for a tool, create the corresponding wrapper class first:
  1. Right-click on the project name in the Solution Explorer.
  2. Open `MFC Class from ActiveX Control`.
  3. Select '`ActivVTools.`<*DesiredTool*>' from the combo box.
  4. Select the interface `_`<*DesiredTool*> and click `->`.
  5. Only select other interfaces if you know that you need them and have not created wrapper classes for them before.
  6. Click `Finish`.
- Now, create the member variable for the tool itself:
  1. Open the form or dialog that contains the tool. (If you have not done so earlier, add the tool by selecting `Insert ActiveX control`).
  2. Right-click on the tool and select `Add Variable`.
  3. Verify that the control ID is set correctly.
  4. Replace the text in the combo box `Variable Type` with the name of the class generated in the last step ('`C`<*DesiredTool*>'). Make sure that the capitalization is correct. You will know when the name matches the existing class, because the text fields `.h file/.cpp file` will gray out.
  5. Enter a name for the variable/
  6. Click `Finish`.

- If you have not done so earlier, append StdAfx.h with the line

```
#include "avtocxtypes.h"
```

  and add `$(AVTROOT)\devenv\dotnet\vcpp\include` to your include directories.

Now you have a member variable for the tool that is of the correct type. If you had let the dialog `Add Variable` create the wrapper class for you, it would not have contained any methods except `Create`. Furthermore, it would be possible to import all classes in a type library when selecting `Add Class`, but they would be derived from `COleDispatchDriver` instead of `CWnd`, which prevents you from using these types as ActiveX control member variables.

To create wrapper for the classes from the ActivVisionTools OCX or from the type library, select `Add Class` and then `MFC Class from Typelib`.

None of the above methods will import user-defined types from the type library, which are necessary, however, for the compilation of some of the wrapper classes. Thus, they are provided in a separate header file. Note that **there is no simple way to use the user-defined types in a Visual C++ program**.

When using debug configurations within Visual C++ 2005, you may experience linker problems regarding embedded manifest files. In this case, please rebuild the `AVTExt` solution yourself.

Note that this section deals with unmanaged C++ applications. For managed C++ see section 1.2.3.2 on page 7, although we recommend to use C# or Visual Basic .NET to create Windows Forms applications.

1.3.6 Additional Information

1.3.6.1 Restrictions

Visual C++ imposes the same restrictions as described for Visual Basic .NET in detail in section 1.2.4.1 on page 9: First, most ActivVisionTools properties must not be changed at design time, because these changes are not saved in the DSC file. Only some properties can only be changed at design time via the property window (see page 9 for a detailed list). Secondly, some dialogs (see page 10 for a list) are only available modally, even if you use the dialog manager as described in section 1.3.3 on page 24.

Besides, Visual C++ currently imposes the restriction that there is no simple way to import the user-defined types from the ActivVisionTools type library to use them in your own program. This applies to all types and enums listed in the **Reference Manual** under `Classes` ▷ `AVTExternalTypes` and `External Types`. How you can at least use enum values as described in section 1.3.6.2.

1.3.6.2 Using ActivVisionTools Enums

Many ActivVisionTools properties use enumeration types; as already mentioned, however, you cannot use these types in your Visual C++ program. But you can at least access and set enum properties by using the corresponding integer values, which are listed in the **Reference Manual**. Please note that the Visual C++ 6.0 wrapper adds a prefix to the corresponding property method. For example, for `AVTView`'s property method `ExecutionMode` the wrapper creates the two methods `GetExecutionMode` and `SetExecutionMode`; note that to set a value it must be passed as a parameter, not via an assignment(=) as in Visual Basic.

In contrast, the Visual C++ .NET wrapper adds the prefixes `get_` and `put_`; thus, for `ExecutionMode` it creates the two methods `get_ExecutionMode` and `put_ExecutionMode`

1.3.6.3 Further Examples

Please have a look in the directory `examples\vcpp` for example projects.

Chapter 2

Creating Customized GUIs

This chapter explains how to create customized graphical user interfaces (*GUIs*). To facilitate this task, the ActivVisionTools family offers tools to access parameters (ActivGUIComponents) and to display selected results (`AVTDecisionResult`). The programming interface provides further useful methods for customization.

2.1 Basics 40

2.2 Example 42

2.2.1 Iconifying Tools 42

2.2.2 Using ActivGUIComponents 44

2.2.3 Displaying Selected Results 46

2.2.4 Extending the GUI via the Programming Interface 47

2.2.5 Further Useful Methods for Customizing a GUI 50

2.1 Basics

ActivVisionTools already provides GUIs via which you can interactively configure your application by setting parameters and then execute it. However, for the final version of an application you'll typically want to customize the overall GUI, i.e., reduce it to those components that are necessary for its use.

Hiding Tools

To use the functionality of an ActivVisionTool, e.g., ActivBarcode, you must place its *master tool*, in the example `AVTBarcode1D`, on the form; its *support tools*, in contrast, need not to be placed on the form but can be opened via the context menu. With the exception of `AVTFeatureCalc` and `AVTAlignment`, all master tools have a GUI containing e.g., combo boxes to select parameter values or buttons to trigger certain actions; in contrast, `AVTFeatureCalc` and `AVTAlignment` appear as icons at design time and are invisible at run time. Via a property, you can "iconify" the other master tools explicitly; for an example see section 2.2.1 on page 42.

Of course, you can also hide a tool by setting its property `Visible` to `'False'`. The disadvantage of this method is that the tool is still visible at design time.

Providing Access to Selected Components

For some applications you might want to hide parts of an ActivVisionTool but still keep some of its GUI components. Components like text boxes, scroll bars, check boxes, or combo boxes are linked to *properties* of a tool, buttons to *methods*; thus, you could use standard Visual Basic elements like combo boxes or buttons and connect them via the programming interface of ActivVisionTools to the properties or methods you need.

Especially in case of the properties, however, this is a cumbersome and error-prone task. To relieve you of this task, the ActivVisionTools family provides ActivGUIComponents: These tools look like standard Visual Basic elements but can be connected to ActivVisionTools properties interactively. Currently, ActivGUIComponents comprise the following tools:

`AVTGUICheckBox` allows to display and modify boolean properties

`AVTGUITextBox` allows to display and modify alphanumeric properties (strings, integers, floating point numbers)

`AVTGUIComboBox` allows to display and modify properties that can adopt a limited set of (string or integer) values

`AVTGUIScrollBar` allows to display and modify numeric properties (integers and floating point numbers)

Section 2.2.2 on page 44 describes an example application in which these tools are used.

Currently, ActivGUIComponents do not provide the equivalent of the Visual Basic button; section 2.2.4 on page 47 shows how to attach standard buttons to the methods of an ActivVisionTool via the programming interface.

Displaying Selected Results

The standard tools for viewing the results of an ActivVisionTools application, i.e., ActivDataView and `AVTDecisionViewResults`, display all results together. This is useful when developing an application; in the final GUI, however, you might want to display only selected results. This can be realized easily using instan). For detailed information regarding `AVTDecisionResult` please refer to the User's Manual for ActivDecision, section 2.5 on page 16. This section also describes another useful tool for displaying the overall results: `AVTDecisionStatus`.

Different GUIs for the Two Execution Modes

By distinguishing between two execution modes, ActivVisionTools allows you to create different GUIs for the configuration and for the actual use of an application. Section 2.2.5 on page 50 describes an example application.

Displaying Your Own Text or Graphical Elements

An ActivVisionTools application automatically displays (the selected) results overlaid on the image in `AVTView`. Via the programming interface, you can add your own elements, e.g., text with `AVTView`'s method `AddOverlayText` or circles with the method `AddOverlayCircle`. PLease refer to the Reference Manual for more details (and more methods).

2.2 Example

In the following, we apply the methods listed in section 2.1 on page 40 step-by-step to customize the GUI of a bar code inspection application. In the corresponding Visual Basic example project `guis\advanced_guis.vbp`, the bar code inspection is already configured, thus you can concentrate on the task of customizing the GUI. For more information about ActivBarcode and ActivDecision please refer to the User's Manual for ActivBarcode and the User's Manual for ActivDecision, respectively.

The project `guis\advanced_guis_final.vbp` shows a possible result of the customization; it contains all the changes described in the following sections.

2.2.1 Iconifying Tools

For most ActivVisionTools, you must place the so-called *master tool* on the form to use the functionality. Exceptions are the *inspection tools* like ActivZoom, which can be accessed in form of dialogs via the context menu of `AVTView`, and display tools like ActivDataView, which can be accessed via the context menus of `AVTView` and of *vision tools* like `AVTBarcode1D`. If the *master tool* is not already represented by an icon only like `AVTFeatureCalc` and `AVTAlignment`, you can iconify it manually.

Please note however, that *master tools* are not available as dialogs (with the exceptions listed above)! You can access their parameters interactively using `AVTViewProperties` (see the User's Manual for ActivView, section 2.3 on page 20 for more information about this tool) or by creating ActivGUIComponents for them as described in section 2.2.2 on page 44. Currently, however, you cannot interactively trigger activities that are provided via buttons, e.g., `AVTBarcode1D`'s `Detect`. For this, you must add a standard button control and link it to the corresponding method of the ActivVisionTools programming interface, which you can find in the **Reference Manual**.

Visual Basic Example

Preparation for the following example:

- ☐ Please open the project `guis\advanced_guis.vbp`, which contains an already configured bar code inspection application.

The following steps are visualized in figure 2.1.

1. At design time, select the tool you want to iconify.
2. In the *Property Window*, set the property `UseIconMode` to `'True'`. Figure 2.2 on page 45 (on the next double page!) shows the effect of iconifying `AVTBarcode1D`.

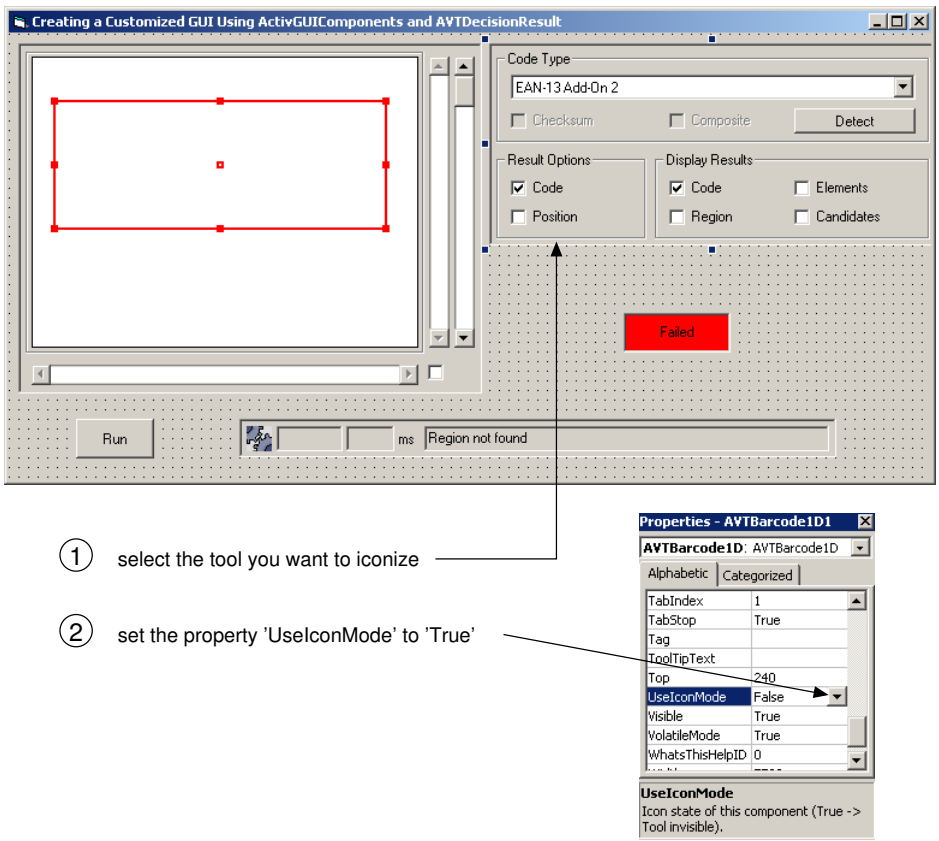


Figure 2.1: How to iconify `AVTBarcode1D` .

Please note that in the current version of ActivVisionTools, some tools (e.g., `AVTDecision`) cannot be iconified, because they does not provide the property `UseIconMode`. Instead, you can set their standard property `Visible` to `'False'`; then, they are invisible at run time, but remain visible in their normal form at design time. This method is used in the project `guis\advanced_guis_final.vbp` to hide `AVTDecision`.

2.2.2 Using ActivGUIComponents

ActivGUIComponents are created very easily: Select the corresponding icon in the Visual Basic toolbox (`AVTGUICheckBox`: , `AVTGUITextBox`: , `AVTGUIComboBox`: , `AVTGUIScrollBar`:) and place it on the form. At design time, you can then customize its appearance; at run time, you select to which property of which ActivVisionTool the component is to be connected.

The following steps are visualized in figure 2.2. In the example, an `AVTGUIComboBox`, an `AVTGUIScrollBar`, and an `AVTGUITextBox` have already added and connected; now, an `AVTGUICheckBox` is to be configured.

① At design time, you can first customize the appearance of the GUI component via the *Property Window*. For an `AVTGUICheckBox`, e.g., you can choose a caption and select how the text is to be aligned; furthermore, you can select between a plain and a graphical style. For an `AVTGUITextBox`, you can select how its content is to be aligned; an `AVTGUIScrollBar` can be oriented horizontally or vertically.

Please note that `AVTGUIComboBox`, `AVTGUIScrollBar`, and `AVTGUITextBox` do not provide a label; the labels in figure 2.2 were created using standard Visual Basic labels.

② You connect the GUI component to an ActivVisionTool property via `AVTViewConnections` (also see the User's Manual for ActivView, section 3.5 on page 36). For this, execute the application and open `AVTViewConnections` by clicking on `AVTView` with the right mouse button and selecting `Connections` in the appearing context menu. Drag the GUI component to the ActivVisionTool to which you want to connect it.

③ Automatically, a context menu pops up and lets you select a property to connect to. If the ActivVisionTool has only one suitable property, the connection is performed automatically.

If you want to change the connection, you can open the context menu by clicking on the GUI component in `AVTViewConnections` with the right mouse button.

④ If the property should only displayed but not be modified, open `AVTViewProperties` by clicking on the GUI component in `AVTViewConnections` with the right mouse button and selecting `Properties`. Here, set the property `ReadOnly` to `'True'`.

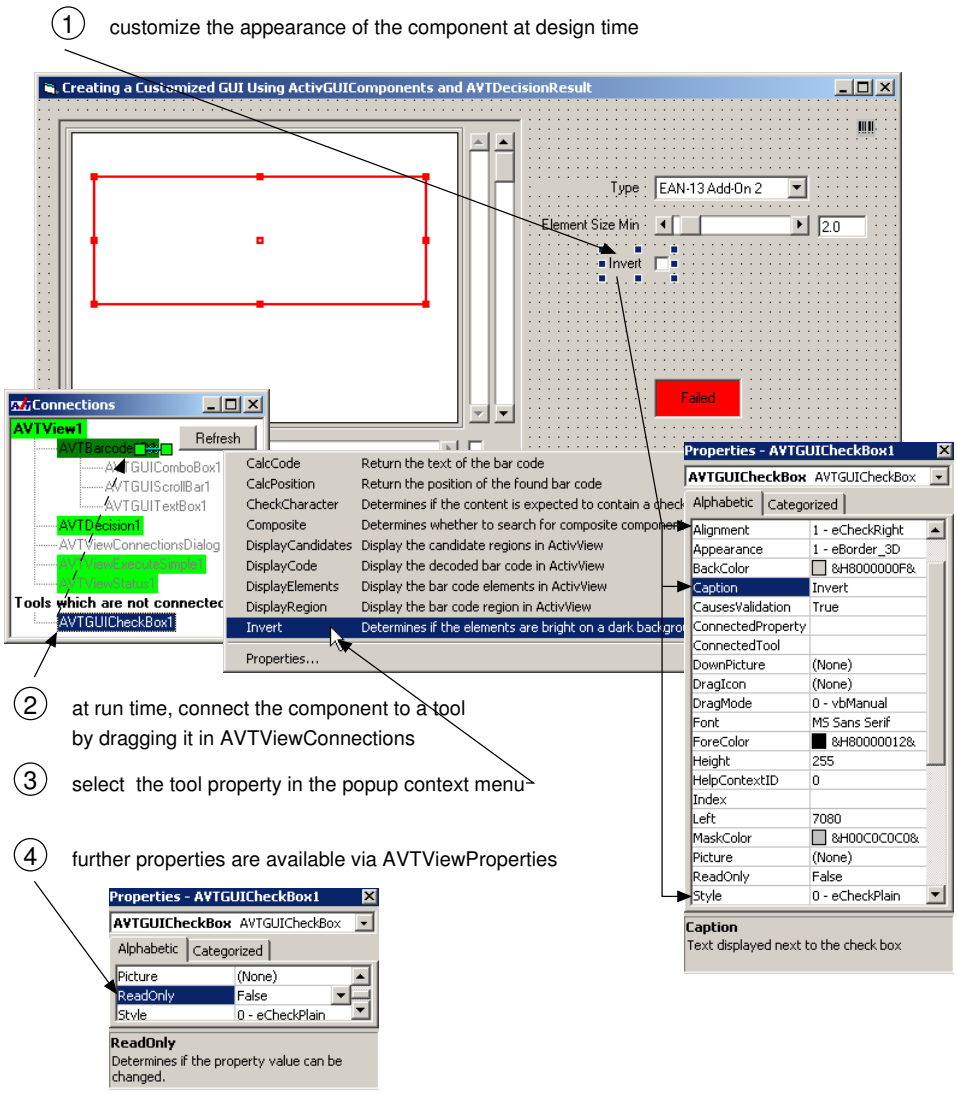


Figure 2.2: Creating and configuring ActivGUIComponents .

2.2.3 Displaying Selected Results Using

You can display selected results, i.e., calculated features and/or their evaluation, using `AVTDecisionResult`. In the Visual Basic toolbox, select and place it on the form. How to configure `AVTDecisionResult` is described in detail in the User's Manual for ActivDecision, section 2.5 on page 16.

In figure 2.3, one instance of `AVTDecisionResult` was created, which displays the extracted bar code and its evaluation. As an exercise, extend the project `guis\advanced_guis.vbp` accordingly; alternatively, start the Visual Basic example project `guis\advanced_guis_final.vbp` which already contains this extension.

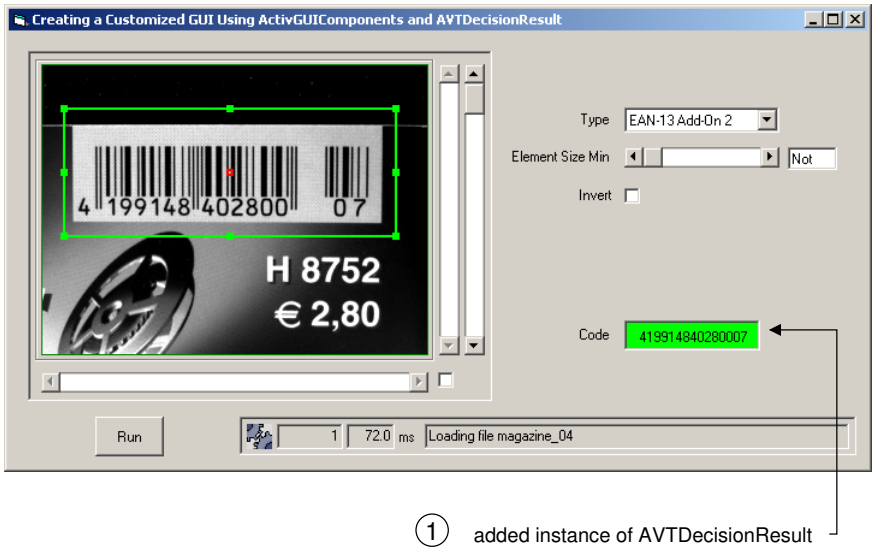


Figure 2.3: Displaying selected results of the bar code inspection.

2.2.4 Extending the GUI via the Programming Interface

In the previous sections, the GUI was customized interactively using dedicated ActivVisionTools, which allowed to access tool properties and to display selected results. For further customization you must resort to programming.

Please note that we assume that readers of this part have at least a basic knowledge of Visual Basic. To work with the programming interface of ActivVisionTools, you must first **add the ActivVisionTools type library to the project's references** by checking the box labeled `ActivVisionTools Type Library` in the Visual Basic menu dialog `Project` ▷ `References`.

Detailed information about the programming interface can be found in the **Reference Manual**.

2.2.4.1 Accessing Tool Methods

As mentioned in section 2.1 on page 40, ActivGUIComponents do not provide the equivalent of a Visual Basic button, which could then be connected to a method of an ActivVisionTool. However, this task can be realized easily via the programming interface. As an example, we show how to create a button to execute the application once: At design time, add a Visual Basic button to the form and configure its name (e.g., `CommandStep`) and caption in the *Property Window*. Then, double-click it; the code window appears with the procedure stub for the handler of the event `Click`. To execute the ActivVisionTools application once, you simply insert a call to the method `ExecuteAllToolsOnce` of `AVTView` (see also chapter 3 on page 53):

```
Private Sub CommandStep_Click()

  Call AVTView1.ExecuteAllToolsOnce(True)

End Sub
```

If you extend the project `guis\advanced_guis.vbp` accordingly, and then start it and click on the button, nothing happens! The reason for this behavior is that the image acquisition is not initialized yet. To automatically initialize it upon starting, add the following line in the event handler `Form_Load` (see section 3.2 on page 59 for more information):

```
Private Sub Form_Load()

  Call AVTView1.InitGrabber(True)

End Sub
```

The Visual Basic example project `guis\advanced_guis_final.vbp` already contains this extension.

2.2.4.2 Accessing Evaluation Parameters

Most parameters available via the GUIs of ActivVisionTools are also accessible in form of properties and can therefore be connected to ActivGUIComponents. A notable exception are the evaluation parameters, which you can modify interactively using `AVTDecisionViewDefaults` and `AVTDecisionViewResults`. In the following, we show how to present only selected evaluation parameters in the GUI via the programming interface. The example goal is to allow to modify the boundaries for the value of the extracted bar code (see figure 2.4). As an exercise, extend the project `guis\advanced_guis.vbp` accordingly; alternatively, start the Visual Basic example project `guis\advanced_guis_final.vbp` which already contains this extension.

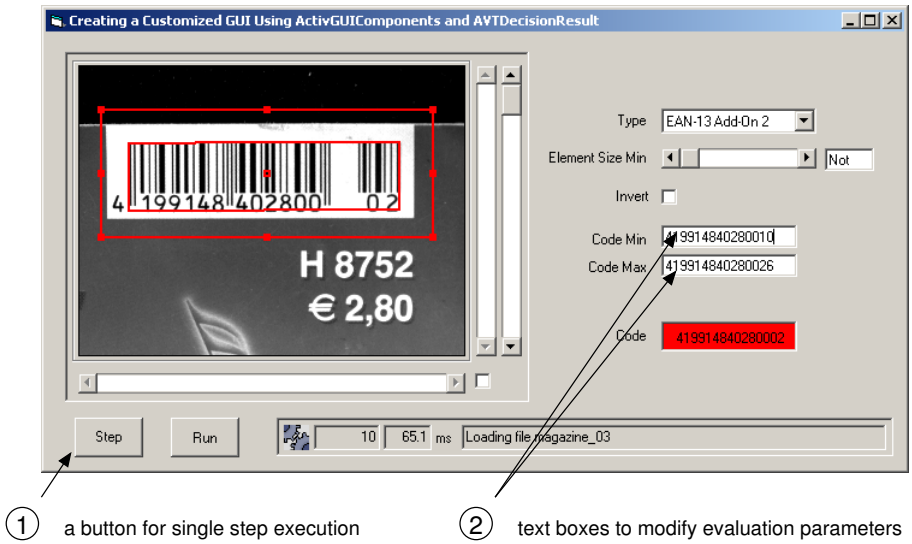


Figure 2.4: Extending the GUI with a button for stepwise execution and with access to evaluation parameters.

First, we add two Visual Basic text boxes to the form. Upon starting the application, these text boxes are to display the current evaluation parameters. These are stored in a global data structure that contains all tool results. To access this data structure, declare a global variable of the class `ActivVTools.AVTApplicationResult`, using the keyword `New` to link it to the global data structure:

```
Dim atAppResults As New ACTIVVTOOLSLib.AVTApplicationResult
```

In the event handler `Form_Load`, the global result data structure already contains the evaluation parameters. To read the current values of the parameters for the evaluation of the bar code, you

first focus on the results of AVTBarcode1D and then call the method GetEvaluator; finally, the read values are entered in the text boxes:

```
Private atBarcodeResults As AVTToolResult
Private handleCode As Integer
Private sCurrentMinCode As Variant
Private sCurrentMaxCode As Variant
Private atInterpret As AVTEvalInterpretationEnum
Private atOp As AVTEvalOperationEnum

Private Sub Form_Load()

  handleCode = AVTBarcode1D1.FeatureHandleCode
  Set atBarcodeResults = atAppResults.ToolResult("AVTBarcode1D1")
  Call atBarcodeResults.GetEvaluator(handleCode, _
                                     sCurrentMinCode, sCurrentMaxCode, _
                                     atInterpret, atOp)
  TextMinCode.Text = sCurrentMinCode
  TextMaxCode.Text = sCurrentMaxCode


End Sub
```

If the content of one of the text boxes is modified, the evaluation parameters are to be modified accordingly. This is realized by creating handlers for the event Change. Below, the handler for the minimum bar code value is shown; the one for the maximum value is constructed analogously. If the new value differs from the stored one, the results of AVTBarcode1D are extracted from the global result data structure, and the new evaluation parameters are set using the method SetEvaluator; finally, all tools are executed once without grabbing to show the effect of the change (see chapter 3 on page 53 about this method).

```
Private Sub TextMinCode_Change()

  Dim sNewMinCode As String

  sNewMinCode = TextMinCode.Text
  If sNewMinCode <> sCurrentMinCode Then
    sCurrentMinCode = sNewMinCode
    Set atBarcodeResults = atAppResults.ToolResult("AVTBarcode1D1")
    Call atBarcodeResults.SetEvaluator(handleCode, _
                                       sCurrentMinCode, sCurrentMaxCode, _
                                       atInterpret, atOp, False)
    Call AVTView1.ExecuteAllToolsOnce(False)
  End If

End Sub
```

2.2.5 Further Useful Methods for Customizing a GUI

By exploiting the fact that ActivVisionTools offers two execution modes, you can let an application present different GUIs for configuring and for running it. For example, you could restrict ActivGUIComponents to the *configuration mode*, and in the *application mode* only display results. As an exercise, extend the project `guis\advanced_guis.vbp` as described below; alternatively, start the Visual Basic example project `guis\advanced_guis_final.vbp` which already contains this extension.

To realize this behavior, you first need access to the property governing the execution mode (`ExecutionMode` of `AVTView`); the easiest way to solve this problem is, of course, to create an instance of `AVTGUIComboBox` and to connect it to the property. Whenever the value of the property changes (i.e., not only if it is changed via the `AVTGUIComboBox`!), the tool raises the event `Change`. Thus, you can place the code for hiding or showing components into the corresponding event handler:

```
Private Sub AVTGUIComboBox2_Change(vNewValue As Variant)

  Dim atMode As AVTExecutionModeEnum

  atMode = AVTGUIComboBox2.ValueIndex
  If atMode = eConfigurationMode Then
    LabelType.Visible = True
    AVTGUIComboBox1.Visible = True
    (...)
  Else
    LabelType.Visible = False
    AVTGUIComboBox1.Visible = False
    (...)
  End If

End Sub
```

The top of figure 2.5 depicts the resulting state of the GUI in the *application mode*.

If you want to prevent the user from changing parameters in the *application mode*, you can disable the popup dialogs by inserting the following lines of code into the event handler `Form_Load`:

```
AVTView1.UsePopUpConfig = True
AVTView1.UsePopUpAppl = False
```

Furthermore, you can allow the editing of ROIs only in the *configuration mode* by inserting the following lines:

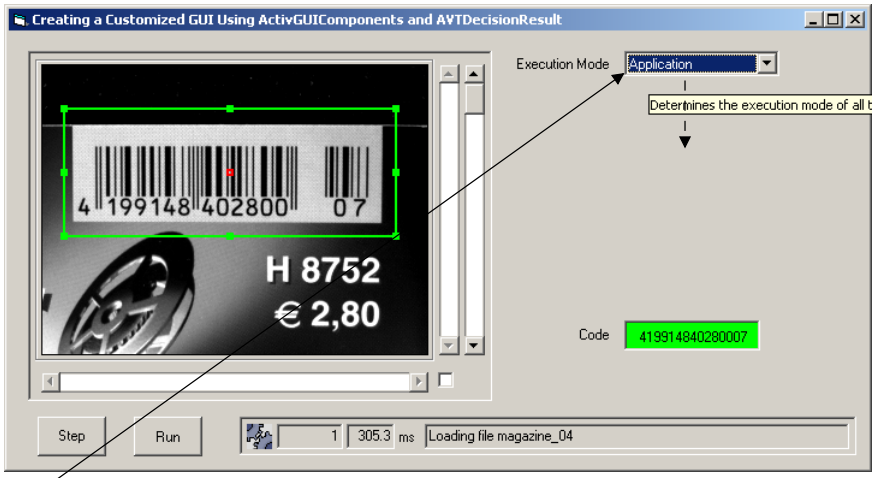


Figure 2.5: Hiding elements.

```
AVTView1.AllowEditROIConfig = True
AVTView1.AllowEditROIAppl = False
```

Chapter 3

Influencing the Execution

ActivVisionTools comes with built-in mechanisms that assure that tools are executed in a suitable sequence and with the necessary input data. This chapter explains how to influence the execution of ActivVisionTools via the programming interface, e.g., how to modify ROIs or how to execute tools in a certain sequence.

3.1 The Standard Execution Cycle 54
3.2 Triggering Execution Cycles 59
3.3 Modifying the Execution Cycle 63
3.3.1 Changing Connections 63
3.3.2 Modifying the Sequence of Execution 63
3.3.3 How to Not Execute a Tool 65
3.4 Modifying the Input Data of Individual Tools 66
3.4.1 Modifying ROIs 66
3.4.2 Modifying the Image 73
3.5 Handling DSC Files via the Programming Interface 83
3.6 Using Multiple Instances of ActivView 88
3.6.1 The Different Modes of Executing Slaves 90
3.6.2 Additional Information About Multiple ActivViews 103

3.1 The Standard Execution Cycle

In this section, we take a closer look at the standard execution cycle, i.e., in which sequence the tools are executed and with which input data. The following sections describe the parts of the ActivVisionTools programming interface that allow to trigger the execution and modify the sequence of execution and the input data and illustrate their use with Visual Basic example projects. Detailed information about the programming interface can be found in the **Reference Manual**.

Note that in the standard execution cycle only one camera is used. Section 3.6 on page 88 shows how the execution cycle is extended for the use of multiple cameras.

One of the features of ActivVisionTools is that in the moment you add a tool to your application, it is automatically connected in a suitable way to other tools. Thus, you can directly start your application, and the tools will be executed in the correct sequence and with the necessary input data, without any further action being required from your part.

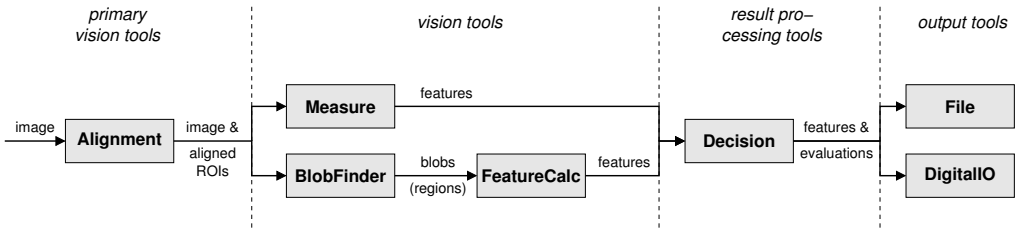


Figure 3.1: An example ActivVisionTools configuration (shortened tool names).

Figure 3.1 depicts an example configuration of ActivVisionTools, including the input data. As can be seen, tools are sorted according to their type: *Primary vision tools* like ActivAlignment are executed first, as their results are necessary for the *vision tools* like ActivMeasure, which are executed next. In fact, currently there are two types of primary vision tools: *calibration tools* like ActivGeoCalib and *change detectors* like ActivAlignment. The *vision tools* calculate so-called *features*, e.g., measure distances or analyze blobs, which are then further processed by *result processing tools* like ActivDecision. Finally, *output tools* like ActivFile output selected results. Note that the figure leaves out one type of tools: the so-called *inspection tools* like ActivHistogram or ActivZoom. These tools are executed at the very end of the cycle.

In figure 3.1, the application contains multiple vision and output tools, respectively, resulting in so-called *branches* in the execution cycle. In which sequence these branches are executed is not predefined; section 3.3.2 on page 63 shows how to influence this sequence via the programming interface.

A branch can also contain a *chain* of tools, e.g., ActivBlobFinder followed by ActivFeatureCalc in figure 3.1. Such chains are always executed as a whole, i.e., ActivFeatureCalc is executed

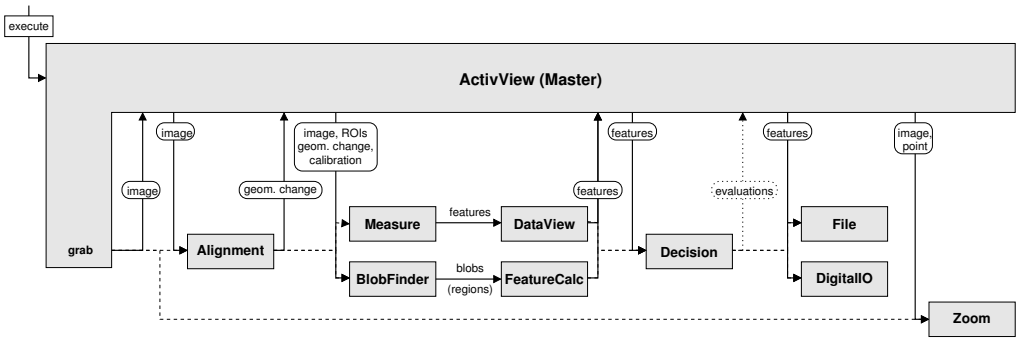


Figure 3.2: Input and output data flow for an example ActivVisionTools configuration (shortened tool names).

directly after ActivBlobFinder. This is also true if the chain contains tools of different types. In figure 3.2, e.g., the vision tool ActivMeasure is followed by the output tool ActivDataView.

In fact, figure 3.1 on page 54 depicts the data flow in an intuitive but not completely correct way: Only within a chain data is passed directly from tool to tool. For the rest, ActivView acts as the central distributor, i.e., it provides the tools with the necessary input and then receives their output data. This is depicted in figure 3.2 for a slightly extended version of the configuration in figure 3.1. We want to remark upon three points regarding the data flow:

- The tools must process the alignment data themselves, i.e., they receive the original ROIs and the alignment data and from this calculate the aligned ROIs.
- The evaluations calculated by ActivDecision do not form new features; instead, they are added to the data structure containing the features. Thus, ActivDecision technically has no output data.
- In figure 3.2, ActivZoom receives input data labeled as 'point'. This denotes the current position of the mouse if you move it over `AVTView`, with the position being given in image coordinates, i.e., in pixels. Such tools are called *point tools*. In general, all *primary vision tools*, *vision tools*, and *inspection tools* may receive this input; among the available ActivVisionTools, currently only ActivZoom does.

Table 3.1 lists for each tool type the input and output data it can receive and create, respectively. Furthermore, it shows the input and output data of the currently available ActivVisionTools, with the exception of ActivGenericInterface, for which you can select the tool type and the input and output data yourself (see chapter 4 on page 107), and ActivGUIComponents, which are not "real" tools but components providing access to properties of the ActivVisionTools (see also section 2.1 on page 40).

Let's take a look at the programming interface of the data flow; how and where to modify the

Type	Input	Output	Tools	Input	Output
			ActivView	none	image, features, point
Calibration Tools	image, point	calibration, features	ActivGeoCalib	image	calibration
Change Detector Tools	image, point	image, geometric change, features	ActivAlignment	image	geometric change, features
Vision Tools	image, regions, XLDs, point	image, regions, XLDs, features, none	ActivMeasure	image	features
			ActivBarcode	image	features
			ActivDataCode	image	features
			ActivBlobFinder	image	regions
			ActivFeatureCalc	regions	features
			ActivOCR	image	features, regions
Result Processing Tools	features	regions, features, none	ActivWordProcess	features	features, regions
			ActivDecision	features	none
Output Tools	features	none	ActivFile	features	none
			ActivDigitalIO	features	none
			ActivSerial	features	none
			ActivDataView	features	none
			ActivFeatureHistogram	features	none
Inspection Tools	image, point	none	ActivHistogram	image	none
			ActivLineProfile	image	none
			ActivZoom	image, point	none

Table 3.1: Possible input and output data for the different tool types and data used by the available ActivVisionTools (except ActivGenericInterface and ActivGUIComponents).

data is discussed in section 3.4 on page 66. You come into contact with the classes containing input and output data whenever you create handlers for the events `Start` and `Finish` of an ActivVisionTool, which are raised when the tool starts and finishes executing, respectively (except ActivGenericInterface, see section 4.3.2 on page 112). For example, the signature of the procedure handling the event `Start` of ActivBlobFinder looks as follows:

```
Private Sub AVTBlobFinder1_Start(atIconicData As _
                                 ActivVTools.AVTImageMessageClass)
```

The class `ActivVTools.AVTImageMessageClass` contains the so-called *iconic data*, i.e., data related to the image to process. Its most relevant components are (see the **Reference Manual** for details about the ActivVisionTools classes):

- `hObject` → `HALCONXLib.HUntypedObjectX`
  the image to process (more exactly: reference to the image)
- `ROIs` → `Collection` of `ActivVTools.AVTROIClass`
  ROIs of the tool that is being executed
- `Affine` → `ActivVTools.AVTAffineClass`
  alignment data
- `Calib` → `ActivVTools.AVTCalibClass`
  calibration data
- `IconicResults` → `Dictionary` of `HALCONXLib.HUntypedObjectX` (key: ROI name)
  iconic objects extracted from the image; this component can contain either regions, e.g., blobs, as described below or so-called *XLDs* (eXtended Line Descriptions) representing, e.g., contour segments (see section 4.4 on page 118 for an example)

After executing, ActivBlobFinder adds its results, i.e., the extracted blobs, to the instance of `ActivVTools.AVTImageMessageClass` and passes this extended version in the event `Finish`:

```
Private Sub AVTBlobFinder1_Finish(atIconicData As _
                                  ActivVTools.AVTImageMessageClass)
```

Typically, an instance of ActivFeatureCalc is connected to an ActivBlobFinder and therefore receives its results, i.e., the iconic data including the extracted blob regions, as input:

```
Private Sub AVTFeatureCalc1_Start(atIconicData As _
                                  ActivVTools.AVTImageMessageClass)
```

For all the blobs passed in the iconic input data, ActivFeatureCalc calculates the selected features, stores them in an instance of the class `ACTIVVTOOLSLib.AVTToolResult`, and passes them on with the event `Finish` (note that Visual Basic displays the corresponding interface class):

```
Private Sub AVTFeatureCalc1_Finish(atToolResults As _
                                   ACTIVVTOOLSLib.IAVTToolResult)
```

Result processing tools like ActivDecision receive the features calculated by one or more tools as input, in form of a `Collection` of `ACTIVVTOOLSLib.AVTToolResult`; like all ActivVisionTools, they also receive the iconic data:

```
Private Sub AVTDecision1_Start(atIconicData As _
                               ActivVTools.AVTImageMessageClass, _
                               cToolResults As Collection)
```

As already noted, ActivDecision does not calculate independent results in form of features, but instead adds its results, i.e., the evaluations, to the data structure containing the features themselves, i.e., the instance of `ACTIVVTOOLSLib.AVTToolResult`, and passes on the augmented results with the event `Finish`:

```
Private Sub AVTDecision1_Finish(atToolResults As Collection)
```

More information about `ACTIVVTOOLSLib.AVTToolResult`, e.g., how to access the features and evaluations, can be found in the User's Manuals for ActivMeasure (see section 4.3 on page 30), ActivBarcode (see section 4.3 on page 26), ActivDataCode (see section 4.3 on page 40), ActivFeatureCalc (see section 4.2 on page 34), ActivOCR (see section 4.3 on page 62), ActivAlignment (see section 4.3 on page 28), and ActivDecision (see section 3.4 on page 30).

Like the *result processing tools*, *output tools* (e.g., ActivSerial) receive the features calculated by one or more tools and the iconic data as input:

```
Private Sub AVTOutputSerial1_Start(atIconicData As _
                                   ActivVTools.AVTImageMessageClass, _
                                   cToolResults As Collection)
```

After performing their output activities, *output tools* pass on the (unmodified) tool results with the event `Finish`:

```
Private Sub AVTOutputSerial1_Finish(atToolResults As Collection)
```

When comparing the event handler signatures of ActivDecision and ActivSerial you might wonder why there exist two different tool types for them. The main reason is that *result processing tools* may calculate their own features (or in the case of ActivDecision calculate evaluations). Therefore, they must be executed before the *output tools* in the standard execution sequence to assure that the latter have access to all calculated features and their evaluations.

As already noted, there exists an additional form of input data: the position of the mouse within `AVTView`. This information is passed in form of the image coordinates, i.e., in pixels, as can be seen in the event `Start` of ActivZoom:

```
Private Sub AVTZoom1_Start(atIconicData As _
                           ActivVTools.AVTImageMessageClass, _
                           iRow As Integer, iCol As Integer)
```

3.2 Triggering Execution Cycles

In this section, we describe how to employ the programming interface to trigger individual execution cycles and to start and stop a continuous execution. To experiment with the presented methods, open the Visual Basic example project `execute\advanced_execution.vbp`. This project does not perform a "real" machine vision task; it just contains multiple ActivVisionTools (ActivMeasure, ActivBarcode, 2 × ActivBlobFinder, 2 × ActivFeatureCalc, ActivDecision), i.e., something to execute. Furthermore, it contains handlers for the event `Start` of all tools, which display a message in a standard Visual Basic list box (marked with (1) in the figure) whenever the tool starts to execute. Figure 3.3 on page 60 depicts the project, already extended as described below.

To execute all tools once, you simply call the method `ExecuteAllToolsOnce` of `AVTView`. This method has two (optional) parameters: With the first, you specify whether a new image is to be grabbed (default value: `'TRUE'`). With the second parameter, you specify whether the so-called *parallel grabbing* mode is to be allowed (default value: `'FALSE'`). In this mode, the image acquisition device already acquires the next image, while the ActivVisionTools application processes the current one (see section 5.1 on page 146 and in the User's Manual for ActivView, section 1.3.3 on page 12, for more information). Please note that the parameter only specifies whether parallel grabbing is *allowed*. Whether this mode is actually used depends on the state of `AVTView`'s property `FGParallelGrabbing` (see section 5.1 on page 146).

If `FGParallelGrabbing` is `'True'` (default), the following line executes all tools with the image that has been grabbed while the previous one was being processed; this is the preferred mode within a real-time application, especially when using an external trigger:

```
Call AVTView1.ExecuteAllToolsOnce(True, True)
```

In contrast, the following line forces the acquisition of a new image before executing all tools; this mode may be useful if the ActivVisionTools application is to control the moment when images are grabbed:

```
Call AVTView1.ExecuteAllToolsOnce(True, False)
```

Finally, the following line executes all tools based on the current image:

```
Call AVTView1.ExecuteAllToolsOnce(False)
```

To experiment with this method, add a button and a check box to the example project (see (2) in figure 3.3). The following code allows to trigger execution cycles by clicking on the button; whether with or without grabbing depends on the value of the check box:

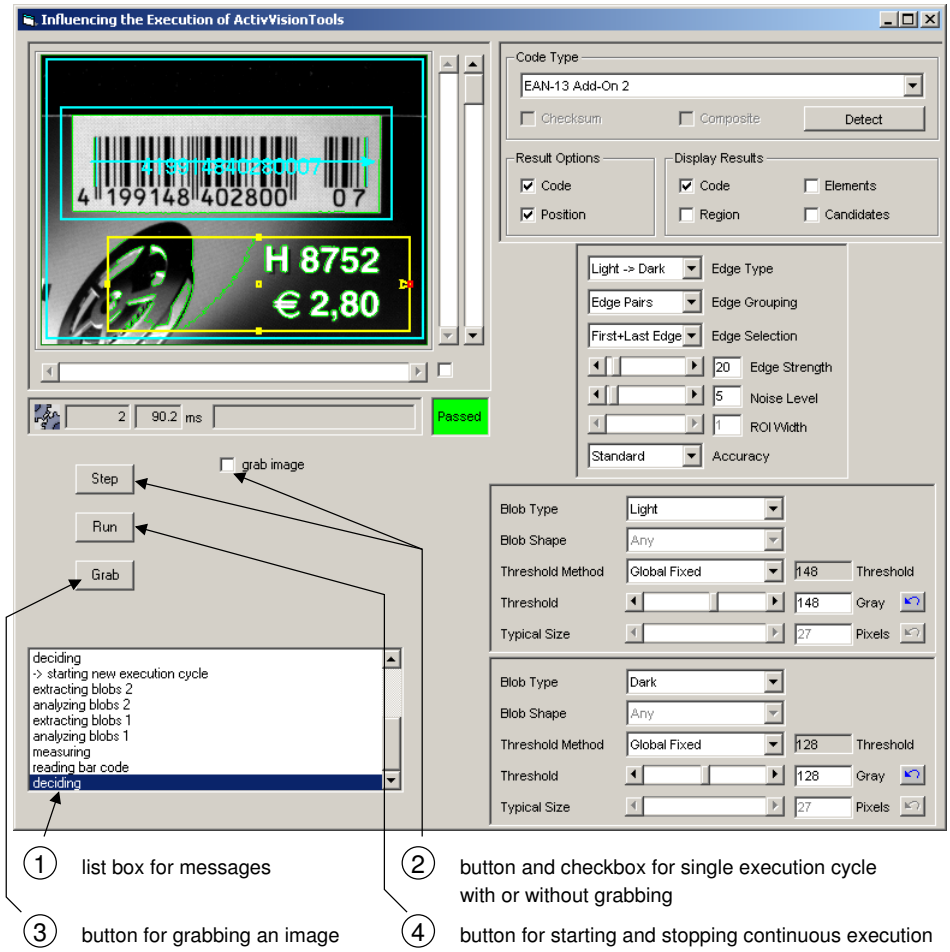


Figure 3.3: The example project for experimenting with the execution of ActivVisionTools, extended to trigger execution cycles.

```
Private Sub CommandExec_Click()

  If CheckGrab.Value = 0 Then
    Call AVTView1.ExecuteAllToolsOnce(False)
  Else
    Call AVTView1.ExecuteAllToolsOnce(True, True)
  End If

End Sub
```

Note that in order for this method to work, the connection to the image acquisition device must already be open (or an image file or sequence must have been loaded). Otherwise, no image is aquired and thus the tools are not executed at all! You can test this behavior by starting your extended project and clicking on the button before loading an image - nothing will happen.

If you already selected suitable parameters in `AVTViewFG` (or if they were stored in the application's DSC file) you can initialize the connection to the image acquisition device (or load the image) via the following method; with the parameter, you can specify whether an image is to be grabbed and processed after connecting:

```
Call AVTView1.InitGrabber(True)
```

If you put this call into the method `Form_Load`, the grabbing is initialize automatically upon starting the project:

```
Private Sub Form_Load()

  Call AVTView1.InitGrabber(True)

End Sub
```

You can also do nothing but grab an image using another method of `AVTView`; this method has an optional parameter specifying whether parallel grabbing is allowed (default: `'TRUE'`).

```
Call AVTView1.GrabImage
```

Note that this method really does nothing but grab an image, in particular, it does not display it. This can be performed using `AVTView`'s method `RefreshDisplay`. Note that this method also displays the tool results; when called after `GrabImage`, it will therefore display the results of processing the previous image. Test this behavior by attaching the following code to another button (③ in figure 3.3 on page 60):

```
Private Sub CommandGrab_Click()

  Call AVTView1.GrabImage
  Call AVTView1.RefreshDisplay

End Sub
```

You can also let your ActivVisionTools application run continuously with the following line:

```
AVTView1.RunState = True
```

This corresponds to a loop in which `ExecuteAllToolsOnce` is called repeatedly; between the calls, Visual Basic events are processed to be able to react to user interactions. Of course, you can also program your own execution loop. Vice versa, you can stop a running application by

setting:

```
AVTView1.RunState = False
```

Note that the application will not stop immediately, but only after the current execution cycle is finished. You can force the application to stop immediately by additionally setting `AVTView`'s property `AbortCycle` to `'True'`; however, you will end up with inconsistent results by doing so in the midst of executing tools. An example for using `AbortCycle` alone can be found in section 3.4.2 on page 73.

With the following code, you can start and stop the application with a single button (④ in figure 3.3 on page 60):

```
Private Sub CommandRunStop_Click()

  If AVTView1.RunState = False Then
    AVTView1.RunState = True
    CommandRunStop.Caption = "Stop"
  Else
    AVTView1.RunState = False
    CommandRunStop.Caption = "Run"
  End If

End Sub
```

The project `execute\advanced_execution_final.vbp` contains the extensions described above, together with others discussed in the following sections.

3.3 Modifying the Execution Cycle

Section 3.1 on page 54 described the standard execution cycle. In this section, we take a closer look at connections between tools and show how to influence the order in which tools are executed.

3.3.1 Changing Connections

For some tools you can decide to which tool they are to be connected. As depicted in figure 3.2 on page 55, most tools are by default connected to ActivView regarding both their input and output, i.e., they receive their input data from and pass their output data back to ActivView. In contrast, ActivFeatureCalc's input cannot be connected to ActivView, because it requires regions (i.e., blobs) as input, which it can get,e.g., from ActivBlobFinder (see table 3.1 on page 56). However, if your application contains more than one instance of ActivBlobFinder, you can decide to which instance ActivFeatureCalc is to be connected.

The basic rule for deciding whether two tools can be connected is simple: The output of the predecessor must include the input type of the successor. However, there are some additional limitations:

- A tool can have multiple successors but only exactly one predecessor.
- ActivDecision, ActivFile, ActivSerial, and ActivDigitalIO can only be connected to ActivView, even if other tools seem to provide the required input (i.e., features). ActivDataView and ActivFeatureHistogram, in contrast, can be connected to feature-producing tools like ActivMeasure, either implicitely by opening them as a dialog via the tool, or explicitly by placing them on the form and then selecting the predecessor as described above.

Note that connections not only determine the data flow, i.e., who provides the input and who receives the output data, but also the sequence of execution, as tool chains are always executed as a whole, independent of the type of the tools that form the chain.

The connections of an ActivVisionTool can be viewed and modified via `AVTViewConnections`, which can be opened by clicking on `AVTView` with the right mouse button and selecting `Connections` in the appearing context menu. Please refer to the User's Manual for ActivView, section 3.5 on page 36, for a detailed description.

3.3.2 Modifying the Sequence of Execution

As described in section 3.1 on page 54, in the standard execution cycle tools are executed in an order depending to their type, except if they are part of a chain. The execution order of tools of the same type, however, does not follow any special rule; in fact, it depends on the

order in which you placed the tools on the form. You can influence the execution order via the so-called *tool descriptor*, which is a component of each ActivVisionTool. The tool descriptor contains a property named `Descriptor.Ordering`. Its default value is `'0'` for all tools. You can choose negative and positive values for it; tools with smaller values will be executed earlier. For example, if an application contains one instance of ActivMeasure, ActivBarcode, and ActivBlobFinder each, the following code leads to the execution sequence ActivBarcode- ActivMeasure- ActivBlobFinder:

```
AVTBarcode1D1.Descriptor.Ordering = -1
AVTBlobFinder1.Descriptor.Ordering = 1
```

This mechanism is useful if you want to use the results of one tool to influence the execution of another, e.g., position a measuring ROI relative to an extracted bar code as shown in section 3.4.1 on page 66, and therefore need to assure that the tools are executed in the correct sequence.

Please note the following:

- `Descriptor.Ordering` **affects only those tools that are connected to ActivView** . This means that if two instances of ActivFeatureCalc are connected to one and the same ActivBlobFinder you cannot influence the sequence in which they are executed!
- In order to influence the sequence of execution of tools of a certain type, you must modify their `Descriptor.Ordering` before the first tool of this type is executed. Later changes only affect the following execution cycles.

Figure 3.4 depicts an extended version of the example project already described in the previous sections. In it, the property `Descriptor.Ordering` of the individual tools can be increased and decreased using scrollbars. The following code realizes this behavior for ActivBarcode:

```
Private Sub VScrollOrderMeasure_Change()

  Dim iOrderMeasure As Integer

  iOrderMeasure = VScrollOrderMeasure.Value
  TextOrderMeasure = iOrderMeasure
  AVTMeasure1.Descriptor.Ordering = iOrderMeasure

End Sub
```

The project `execute\advanced_execution_final.vbp` contains the extensions shown in figure 3.4, together with others discussed in the following section.

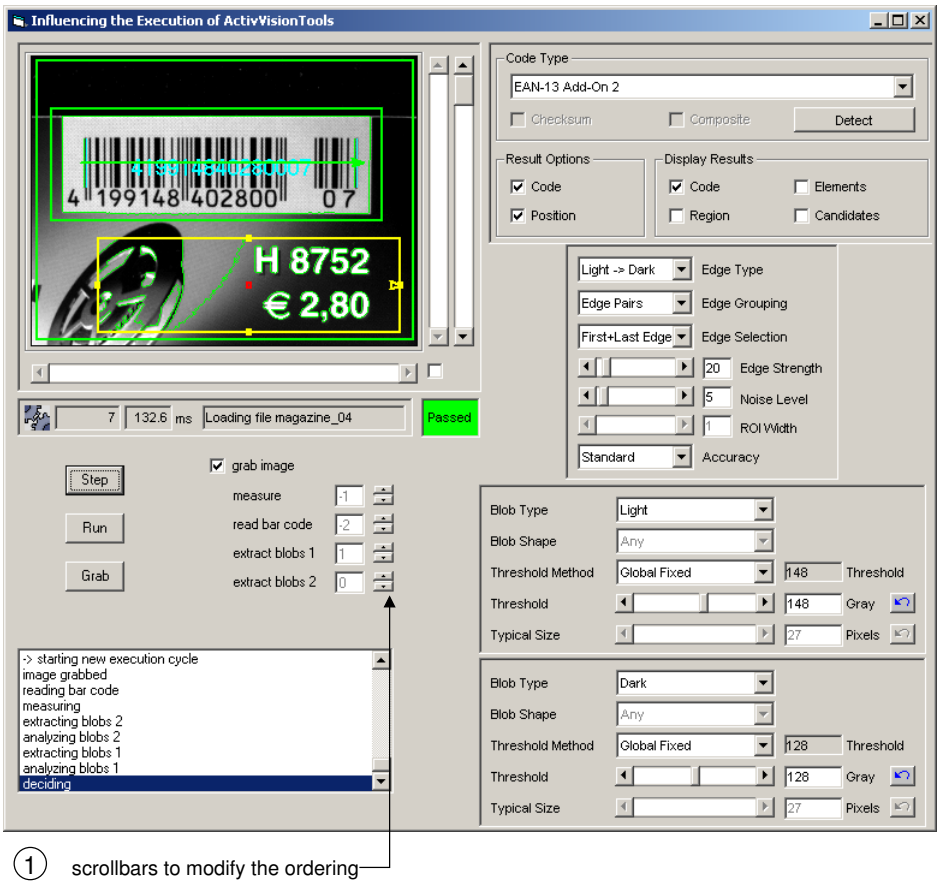


Figure 3.4: The example project for experimenting with the execution of ActivVisionTools, extended to modify the sequence of execution.

3.3.3 How to Not Execute a Tool

In some application you might wish to execute a tool not in all cycles but only “on demand”. This can be realized by connecting it to a slave instance of `AVTView` in the mode `'Independent'`. Please refer to section 3.6.1.4 on page 98 for more information.

3.4 Modifying the Input Data of Individual Tools

The previous section showed how to modify the execution cycle, i.e., whether tools are executed at all, from which tool they receive their input data, and in which sequence the tools (or tool chains) are executed. In this section, we go one step further and show how to influence the execution of a tool by modifying its input data.

! Please note that for the current version of ActivVisionTools, we recommend to **leave the calibration and alignment data, the iconic results, and the feature results and evaluations untouched!** In the following, we therefore only discuss how to modify the input image and the ROIs of a tool.

3.4.1 Modifying ROIs

As described in section 3.1 on page 54, a tool receives its ROIs as part of its *iconic input data* in form of a `Collection` of `ActivVTools.AVTROIClass` (see the property `ROIs` of the class `ActivVTools.AVTImageMessageClass` in the **Reference Manual**). For the current version

! of ActivVisionTools, however, we recommend to **leave this property untouched**. Instead, you can modify, create, and delete ROIs via methods of `AVTView`, which is in control of all ROIs of all ActivVisionTools.

The following code, e.g., creates a rectangular ROI for ActivBarcode over almost the complete image:

```
Call AVTView1.CreateROIRectangle1("AVTBarcode1D1", _
                                  10, 10, AVTView1.HeightImage - 10, _
                                  AVTView1.WidthImage - 10)
```

ActivView assigns a name to this ROI the same way as when you create the ROI interactively via `AVTViewROI`. As an alternative, you can also specify the name in an (optional) parameter of the methods for creating ROIs; the following line, e.g., create a line-shaped ROI for ActivMeasure with the name `"MyROI"`:

```
Call AVTView1.CreateROILine("AVTMeasure1", _
                            100, 100, 100, 300, 10, "MyROI")
```

Yet, what happens if an ROI with this name already exists? If it belongs to the same tool and has the same shape, the existing ROI is modified; if it belongs to another tool or has a different shape, the existing ROI is deleted and a new one is created under the same name. Thus, with the following code the ROI created earlier for ActivMeasure would be deleted and replaced by a new ROI for ActivBarcode:

```
Call AVTView1.CreateROIRectangle1("AVTBarcode1D1", _
                                  100, 100, 400, 400, "MyROI")
```

Therefore, if you plan to modify an ROI, you should store the reference to it, which is actually returned by the creation methods:

```
Dim atMeasureROI2 As AVTROIClass
Set atMeasureROI2 = AVTView1.CreateROILine("AVTMeasure1", _
                                           200, 100, 200, 300, 10)
```

Then, you can access the name of the ROI and move it safely afterwards:

```
Call AVTView1.CreateROILine("AVTMeasure1", _
                            300, 100, 300, 300, 10, atMeasureROI2.Name)
```

But how get the reference if the ROI has already been created, e.g., because it was stored in the application's DSC file by a former execution? If you know the name of the ROI you can get the reference to it via a corresponding method of ActivView; however, in our case we need the reference to get at the name, therefore this way is blocked. Instead, we exploit the fact that each tool stores reference to its ROIs in its so-called *registration*, which contains, besides other information, a property named `Registration.ROIs`, which in turn contains the ROIs of the tool in form of a `Collection` of `ActivVTools.AVTROIClass`. Thus, the following code accesses the second ROI of ActivBarcode:

```
Dim atBarcodeROI2 As AVTROIClass
Set atBarcodeROI2 = AVTBarcode1D1.Registration.ROIs.Item(2)
```

Besides the name, you can also access the parameters of an ROI and use this information, e.g., to move it relative to to its current position:

```
Dim atROIData As AVTROIRectangle1Type
atROIData = atBarcodeROI2.ROIRectangle1
Call AVTView1.CreateROIRectangle1("AVTBarcode1D1", _
                                  atROIData.Row1 - 50, atROIData.Col1 - 50, _
                                  atROIData.Row2 - 50, atROIData.Col2 - 50, _
                                  atBarcodeROI2.Name)
```

ActivView provides more methods, e.g., to create a copy of an ROI, to delete an ROI by name or to delete all ROIs. Please refer to the **Reference Manual** for more information.

There are some points to keep in mind when creating or modifying ROIs via the programming interface within the ActivVisionTools execution cycle:

1. Changes regarding the ROIs of a tool must be effected before the tool is executed.
2. All changes are permanent, i.e., they are not reset at the end of an execution cycle.

Before we present example applications of the methods to create and modify ROIs, we'd like to digress briefly into the workings of ActivAlignment. Even if the display in `AVTView` seems to suggest the opposite, ActivAlignment does not move the ROIs of other tools, but only provides the information about the movement of the anchor in form of the already mentioned property `Affine` of the iconic data class `ActivVTools.AVTImageMessageClass`, which also contains the ROIs of a tool. Its the tools themselves who combine this information to work on an aligned version of their ROIs. Thus, when using ActivAlignment you can still create and modify ROIs via the programming interface, but remember that thereby you specify their unaligned position and orientation.

3.4.1.1 Example 1: Aligning a Line of Measurement to an Extracted Bar Code

We want to continue with the example Visual Basic project presented in section 3.2 on page 59 and extend it such that the line of measurement is automatically placed over the extracted bar code. In other words, we want to align the ROI of ActivMeasure to the result of ActivBarcode. Figure 3.5 shows the result; you can find the necessary extensions in the project `execute\advanced_execution_final.vbp`, together with others discussed in the following section.

First, we declare and initialize global variables for the reference to the ROI of ActivMeasure to align and for the features handles for the results of ActivBarcode:

```
Private atROIMeasure As AVTROIClass
Private handleCodeRow As Integer, handleCodeCol As Integer
Private handleCodeAngle As Integer, handleCodeWidth As Integer
Private handleCodeHeight As Integer

Private Sub Form_Load()

  handleCodeRow = AVTBarcode1D1.FeatureHandleRow
  handleCodeCol = AVTBarcode1D1.FeatureHandleColumn
  handleCodeAngle = AVTBarcode1D1.FeatureHandleOrientation
  handleCodeWidth = AVTBarcode1D1.FeatureHandleMajorExtent
  handleCodeHeight = AVTBarcode1D1.FeatureHandleMinorExtent
  Set atROIMeasure = AVTMeasure1.Registration.ROIs(1)

End Sub
```

Obviously, the earliest moment to align the ROI of ActivMeasure to the bar code is right after ActivBarcode has finished to extract the bar code, i.e., in a handler for its event `Finish`. By setting the property `Descriptor.Ordering` of ActivBarcode to `'-1'` (see figure 3.5), we assure that ActivMeasure is executed after ActivBarcode.

The following code serves to read the position and orientation of the extracted bar code (see the

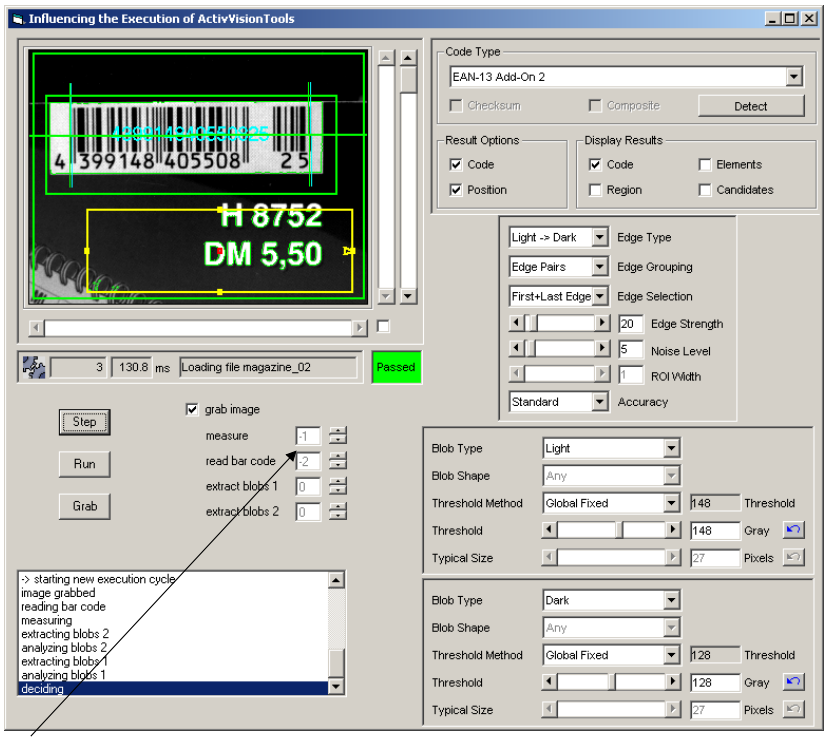


Figure 3.5: The example project for experimenting with the execution of ActivVisionTools, extended to place the line of measurement over the extracted bar code.

User's Manual for ActivBarcode, section 4.3.1 on page 27, for more information about accessing its results via the programming interface); note that the alignment is restricted to the *application mode*, to allow to set up the bar code extraction successfully in the *configuration mode*:

```
Private Sub AVTBarcode1D1_Finish(atToolResults As _
                                 ACTIVVTOOLSLib.IAVTToolResult)

  Dim atROIResult As AVTROIResult
  Dim dCodeCenterRow As Double, dCodeCenterCol As Double
  Dim dCodeAngle As Double, dCodeWidth As Double
  Dim dCodeHeight As Double

  If AVTView1.ExecutionMode = eConfigurationMode Then
    Exit Sub
  End If

  Set atROIResult = atToolResults.ROIResult(atToolResults.ROIResultNames(0))
  dCodeCenterRow = atROIResult.ObjectValue(handleCodeRow, 0)
  dCodeCenterCol = atROIResult.ObjectValue(handleCodeCol, 0)
  dCodeAngle = atROIResult.ObjectValue(handleCodeAngle, 0)
  dCodeWidth = atROIResult.ObjectValue(handleCodeWidth, 0)
  dCodeHeight = atROIResult.ObjectValue(handleCodeHeight, 0)
```

Based on this, the following code computes the new parameters of the line of measurement to place it over the bar code (also see figure 3.6 on page 71):

```
  Dim dHalfMeasureWidth As Double, dAngleInRad As Double
  Dim dCosAngle As Double, dSinAngle As Double
  Dim dMeasureRow1 As Double, dMeasureCol1 As Double
  Dim dMeasureRow2 As Double, dMeasureCol2 As Double

  dHalfMeasureWidth = dCodeWidth / 2 + 10
  dAngleInRad = DegToRad(dCodeAngle)
  dCosAngle = Cos(dAngleInRad)
  dSinAngle = Sin(dAngleInRad)
  dMeasureRow1 = dCodeCenterRow + dSinAngle * dHalfMeasureWidth
  dMeasureRow2 = dCodeCenterRow - dSinAngle * dHalfMeasureWidth
  dMeasureCol1 = dCodeCenterCol - dCosAngle * dHalfMeasureWidth
  dMeasureCol2 = dCodeCenterCol + dCosAngle * dHalfMeasureWidth
  Call AVTView1.CreateROILine("AVTMeasure1", _
                              dMeasureRow1, dMeasureCol1, dMeasureRow2, _
                              dMeasureCol2, dCodeHeight * 0.5, _
                              atROIMeasure.Name)

End Sub
```

Note that the orientation of the bar code is specified in degrees; the following procedure allows to convert it to radians:

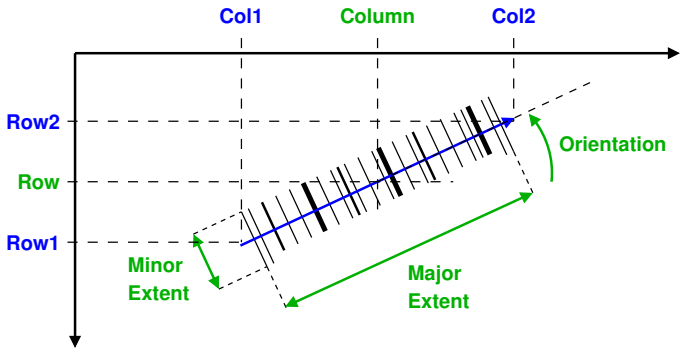


Figure 3.6: Features of ActivBarcode and parameters of a line-shaped ROI .

```
Private Function DegToRad(dAngle As Double)
  DegToRad = dAngle / 180# * 3.14159265358979
End Function
```

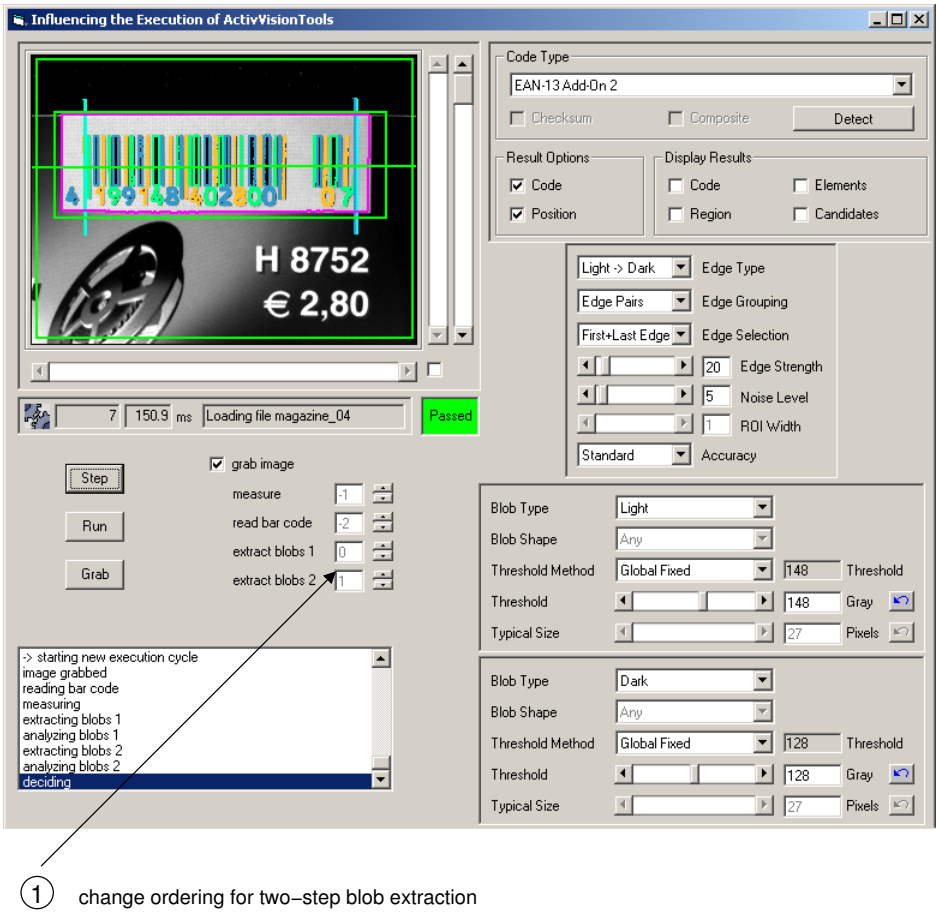


Figure 3.7: The example project for experimenting with the execution of ActivVisionTools, extended to extract the elements of the bar code in two steps.

3.4.1.2 Example 2: Realizing a Two-Step Blob Extraction

Another useful application of modifying an ROI of a tool based on the results of another is to extract blobs in two steps. As an example, we extend the by now well-known Visual Basic project presented in the previous sections to extract the dark elements of the bar code. If one does not know the rough position of the bar code, i.e., if the ROI for ActivBlobFinder must cover the complete image, it is difficult if not impossible to find parameters to extract only the desired blobs. However, if one could reduce the ROI to the white region around the bar code, extracting the elements would present no problem whatsoever. The solution to this task is, therefore, to extract

the white region around the bar code first, and then pass this region to a second ActivBlobFinder. Analogously to the first example, we assure that the second ActivBlobFinder is executed after the first via the property `Descriptor.Ordering`. Figure 3.7 shows the result; you can find the necessary extensions in the project `execute\advanced_execution_final.vbp`.

In the handler for ActivBlobFinder's event `Finish`, we first access the result of the first blob extraction, i.e., the extracted white blob region. Note that this activity is restricted to the *application mode*, to allow to set up the extraction of the white region successfully in the *configuration mode*:

```
Private Sub AVTBlobFinder1_Finish(atIconicData As _
                                  ActivVTools.AVTImageMessageClass)

  Dim sROI1Name As String
  Dim hWhiteRegion As HUntypedObjectX

  If AVTView1.ExecutionMode = eConfigurationMode Then
    Exit Sub
  End If

  sROI1Name = AVTBlobFinder1.Registration.ROIs(1).Name
  Set hWhiteRegion = atIconicData.IconicResults(sROI1Name)
```

The extracted region is then transformed into an ROI for the second blob extraction:

```
  Call AVTView1.CreateROIRegion("AVTBlobFinder2", hWhiteRegion, "WhiteRegion")

End Sub
```

An alternative solution to this task using multiple instances of `AVTView` is presented in section 3.6.1.2 on page 94.

3.4.2 Modifying the Image

First of all, we want to distinguish between two ways of "modifying the image": You can either

1. use another image or
2. modify the image data, i.e., the pixel values.

The first way may seem unnecessary, but it is very important, actually. When using RGB color images, e.g., tools like ActivMeasure, ActivHistogram, or ActivAlignment by default process only the red channel (see the User's Manual for ActivView, section 1.3.2 on page 10, for more information about color images). To process another channel, you just split the original RGB

image into three individual images corresponding to the three color channels and then use the desired image when executing a tool. This case is described below. As described in section 5.2 on page 147, a similar method allows to process images grabbed simultaneously from multiple cameras.

Modifying the image data typically means preprocessing the image, e.g., smoothing it. Another application, which is described below, is to convert an RGB image to another color space for easier processing. Please note that – with the exception of 3D image rectification (see the User's Manual for ActivGeoCalib, section 4.2 on page 26) – **images can only be modified using HALCON,** MVTec's unique machine vision library, which is the base of ActivVisionTools. To use HALCON operators within your ActivVisionTools application, you need a license for the (virtual) tool ActivHALCONFoundation.

In fact, most HALCON operators do not modify the data of their input image but create a new one containing the modified gray values. Thus, in this case you will automatically combine the two ways of modifying an image, i.e., use another image, which contains modified image data.

In the following, we list the points in the ActivVisionTools execution cycle at which you can modify the image, and how long the modification will "live" each time, i.e., which parts of the rest of the cycle are affected by it. Before we start, we want to point out a potential source of confusion: The data type used for passing the image, i.e., `HALCONXLib.HUntypedObjectX` (see section 3.1 on page 54, does not contain the image data, actually, but only the ID under which the image is stored in HALCON's iconic database. In the following, we use the term "image ID" when speaking of a `HALCONXLib.HUntypedObjectX`.

① **After image acquisition**
After grabbing a new image, its ID is stored in `AVTView`'s property `HalconImage`. Immediately afterwards, `AVTView` raises the event `ImageGrabbed` and passes (a reference to) `HalconImage` as a parameter. Thus, the first possible moment to access the image is within a handler for this event:

```
Private Sub AVTView1_ImageGrabbed(hImage As HALCONXLib.HUntypedObjectX, _
                                  ByVal lError As Long)
```

If you modify `AVTView`'s property `HalconImage` in the event handler, the complete execution cycle is affected, i.e., all tools receive the modified image, including the display at the end of the cycle, which uses the image ID stored in `HalconImage`. At this point, there is no difference between the two ways of modifying an image, i.e., you will get the same effect by using another image, i.e., by changing the image ID, as you get by modifying the image data directly.

② **Before the processing starts**
As already described in section 3.1 on page 54, all tools receive their iconic input data in form of an instance of the class `ActivVTools.AVTImageMessageClass` (see also the **Reference Manual**). Before the first tool is executed, this instance of `Ac-`

tivVTools.AVTImageMessageClass is initialized with the current iconic data; the property hObject is set to the image ID currently stored in AVTView's property HalconImage. Then, AVTView raises the event ProcessingStart, passing the instance of ActivVTools.AVTImageMessageClass as a parameter:

```
Private Sub AVTView1_ProcessingStart(atGlobalIconicData As _
                                     ActivVTools.AVTImageMessageClass)
```

If you modify the property hObject within the event handler, all tools receive the modified image as part of their input. In contrast to above, at this there is a significant difference between the two ways of modifying the image: If you use another image, all tools process this image, yet the original image is displayed at the end, because HalconImage still contains the original image's ID. On the other hand, if you modify the image data directly, hObject and AVTView.HalconImage still contain the same image ID, i.e., the latter now also points to the modified image, thus not only do the tools process the modified image, but it it is also displayed.

Of course, you could also modify AVTView.HalconImage within the event handler. Again, you get different results for the two ways of modifying: If you use another image, i.e., change the image ID, the tools process the original image, but the modified image is displayed. If, on the other hand, you modify the image data directly, hObject still points to the now modified image, which is thus not only displayed but also processed.

③ **When a tool starts to execute**

As described above, a tool receives the ID of the image to process in form of the property hObject of the class ActivVTools.AVTImageMessageClass. Before executing, the tool raises the event Start, passing the instance of ActivVTools.AVTImageMessageClass as parameter, like ActivBlobFinder:

```
Private Sub AVTBlobFinder1_Start(atIconicData As _
                                 ActivVTools.AVTImageMessageClass)
```

If you modify the image in the event handler, the tool will process the modified image. Furthermore, if the tool is part of a chain, it will pass on the modified image to its successor(s). Note that for ActivGenericInterface a slightly different mechanism is used; please refer to chapter 4 on page 107 for details.

At the end of a chain, hObject is restored to the value it had after the event ProcessingStart, i.e., the next tool to execute again receives the original image ID. If, however, you modified the image data directly, the restoration of the image ID has no effect, i.e., the rest of the execution cycle processes the modified image; furthermore, if AVTView.HalconImage still contains the same image ID as hObject, the modified image is also displayed.

The effect of modifying AVTView.HalconImage within the handler for the event Start is similar to that of modifying it within the handler for the event ProcessingStart

described above: If you change the image ID, the modified image is displayed, but the original one is processed. If, on the other hand, you modify the image data directly while `hObject` still contains the same image ID as `AVTView.HalconImage`, rest of the execution cycle processes and displays the modified image.

To sum up this lengthy description, we recommend the following:

- **Never modify the image data directly**, but create a new image containing the modified data and then use its ID. Thus, you can easily keep track of which image is processed by which tool.
- If images must be modified generally, i.e., if the complete execution cycle is to process (and display) the modified image, modify `AVTView.HalconImage` within a handler for `AVTView`'s event `ImageGrabbed`. On the other hand, if only certain tools need a modified image, modify `ActivVTools.AVTImageMessageClass.hObject` within a handler for the tool's event `Start`.
- To select another image for display, modify `AVTView.HalconImage` within a handler for `AVTView`'s event `ProcessingStart`.

Below, we present two examples for modifying the image, both using color images. The first shows how to process not only the default, i.e., red channel, but also another one. The second one uses HALCON to transform the RGB image into another color space that facilitates the extraction of color blobs.

3.4.2.1 Example 1: Processing Different Channels of an RGB Image

As already noted, in the current version ActivVisionTools can grab color images, but they process only one channel, by default the red one. Using the methods described above, however, you can let a tool process a different channel. In the Visual Basic example project `color\advanced_color_split.vbp`, this is used to extract different parts of a board based on their color. The project is already completely configured, including the Visual Basic code described below. It uses two instances of ActivBlobFinder: One extracts the (red) resistors, which show up in the red channel as bright regions, while the other extracts the (bluish) capacitors, which show up as bright regions in the blue channel (see figure 3.8).

Note that to create instances of the class `HALCONXLib.HUntypedObjectX` or to set variables or parameters belonging to this class no HALCON license is required; however, you must **add the HALCON/COM library to the project's references** by checking the corresponding box in the Visual Basic menu dialog `Project` ▷ `References`.

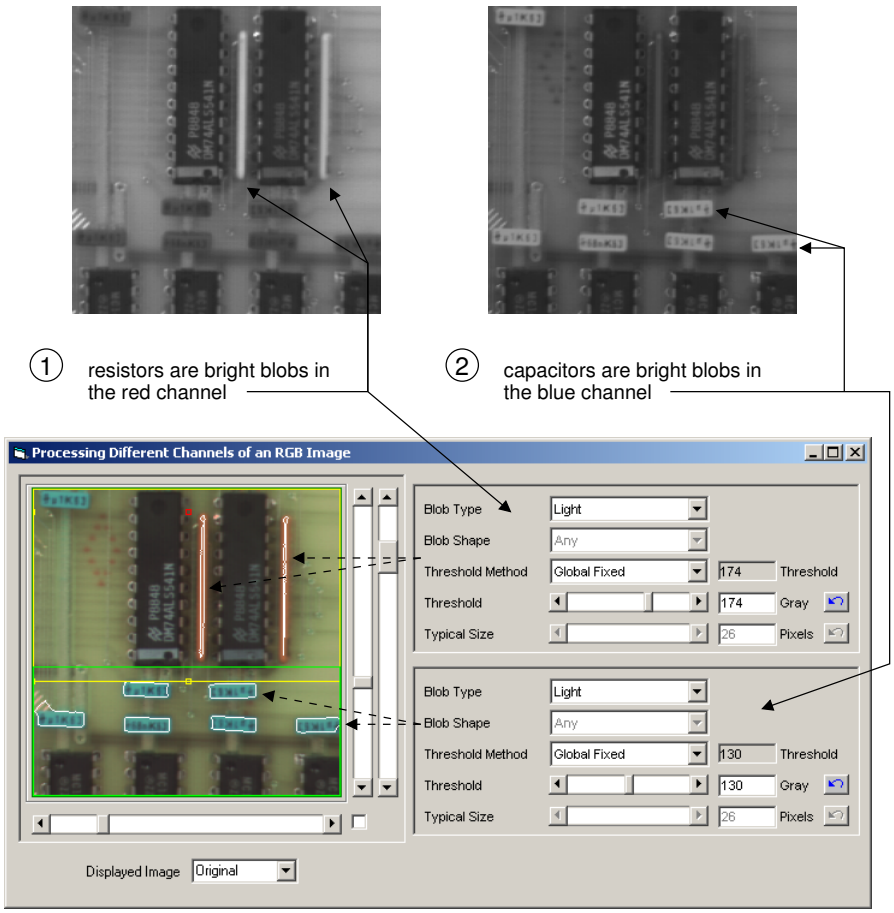


Figure 3.8: Extracting different objects in different channels of an RGB image.

To be able to let tools process a selected channel, we first split the original RGB into its three channels and store their image IDs in global variables. The first possible moment to do this is, of course, in a handler for `AVTView`'s event `ImageGrabbed`. The following code realizes this task, including a safety check whether the original image is indeed a three-channel image:

```
Private imageOriginal As HUntypedObjectX, imageRed As HUntypedObjectX
Private imageGreen As HUntypedObjectX, imageBlue As HUntypedObjectX

Private Sub AVTView1_ImageGrabbed(hImage As HALCONXLib.HUntypedObjectX, _
                                  ByVal lError As Long)

  Set imageOriginal = hImage

  If AVTView1.HalconImageChannelCount = 3 Then
    Set imageRed = AVTView1.HalconImageChannel(1)
    Set imageGreen = AVTView1.HalconImageChannel(2)
    Set imageBlue = AVTView1.HalconImageChannel(3)
  Else
    Set imageRed = hImage
    Set imageGreen = hImage
    Set imageBlue = hImage
  End If

End Sub
```

To let the second instance of ActivBlobFinder process the blue channel, you simply change its input image accordingly in a handler to its event `Start`:

```
Private Sub AVTBlobFinder2_Start(atIconicData As _
                                 ActivVTools.AVTImageMessageClass)

  Set atIconicData.hObject = imageBlue

End Sub
```

An alternative realization of this task using multiple instances of `AVTView` is described in section 3.6.1.3 on page 96.

The example project contains additional code that allows to select the displayed image interactively via a combo box (see figure 3.8 on page 77). Following the recommendations given above, this is realized by changing `AVTView.HalconImage` within a handler for `AVTView`'s event `ImageGrabbed`:

```
Private Enum DisplayedImage
  eOriginal = 0
  eRed = 1
  eGreen = 2
  eBlue = 3
End Enum

Private Sub AVTView1_ProcessingStart(atGlobalIconicData As _
                                     ActivVTools.AVTImageMessageClass)

  Select Case ComboDisplayedImage.ListIndex
    Case eOriginal: Set AVTView1.HalconImage = imageOriginal
    Case eRed: Set AVTView1.HalconImage = imageRed
    Case eGreen: Set AVTView1.HalconImage = imageGreen
    Case eBlue: Set AVTView1.HalconImage = imageBlue
  End Select

End Sub
```

The following code initializes the combo box:

```
Private Sub Form_Load()

  Call ComboDisplayedImage.AddItem("Original", eOriginal)
  Call ComboDisplayedImage.AddItem("Red", eRed)
  Call ComboDisplayedImage.AddItem("Green", eGreen)
  Call ComboDisplayedImage.AddItem("Blue", eBlue)
  ComboDisplayedImage.ListIndex = eOriginal

End Sub
```

3.4.2.2 Example 2: Color Blob Extraction in the Color Space HSI

In the example presented above, we managed to extract differently colored objects from the different channels of an RGB image. However, the color space RGB is not the best-suited one for color image processing. An alternative is the color space HSI, whose three channels represent the hue, saturation, and intensity information. The hue image is well-suited to extracting differently colored blobs, while the intensity image is a good choice for ActivMeasure. The Visual Basic example project `color\advanced_color.vbp` allows to experiment with this color space (see figure 3.9). It also contains an instance of ActivHistogram, thus you can examine the distribution of pixel values in the different channels; as can be seen in the figure, in the hue channel different colors show up as clearly separated peaks, that's why this channel is so well-suited to blob extraction.

Note however, that the example uses HALCON to transform the RGB image into an HSI image, thus **you need a license for the (virtual) tool ActivHALCONFoundation to run this project** (or a HALCON license).

As in the previous example, the channels of the RGB image are accessed within a handler for `AVTView`'s event `ImageGrabbed`. The channels are then transformed into the channels of an HSI image using the HALCON operator `TransFromRgb`. The operator is called via a global variable of the class `HALCONXLib.HOperatorSetX`:

```
Private hOpSet As New HOperatorSetX

Private imageOriginal As HUntypedObjectX
Private imageRed As HUntypedObjectX
Private imageGreen As HUntypedObjectX
Private imageBlue As HUntypedObjectX
Private imageHue As HUntypedObjectX
Private imageSaturation As HUntypedObjectX
Private imageIntensity As HUntypedObjectX
Private imageHueHighSat As HUntypedObjectX

Private Sub AVTView1_ImageGrabbed(hImage As HALCONXLib.HUntypedObjectX, _
                                  ByVal lError As Long)

  Set imageOriginal = hImage
  Set imageRed = AVTView1.HalconImageChannel(1)
  Set imageGreen = AVTView1.HalconImageChannel(2)
  Set imageBlue = AVTView1.HalconImageChannel(3)

  Call hOpSet.TransFromRgb(imageRed, imageGreen, imageBlue, _
                           imageHue, imageSaturation, imageIntensity, "hsi")
```

Then, further HALCON operators are called to limit the so-called *domain* of the hue image to

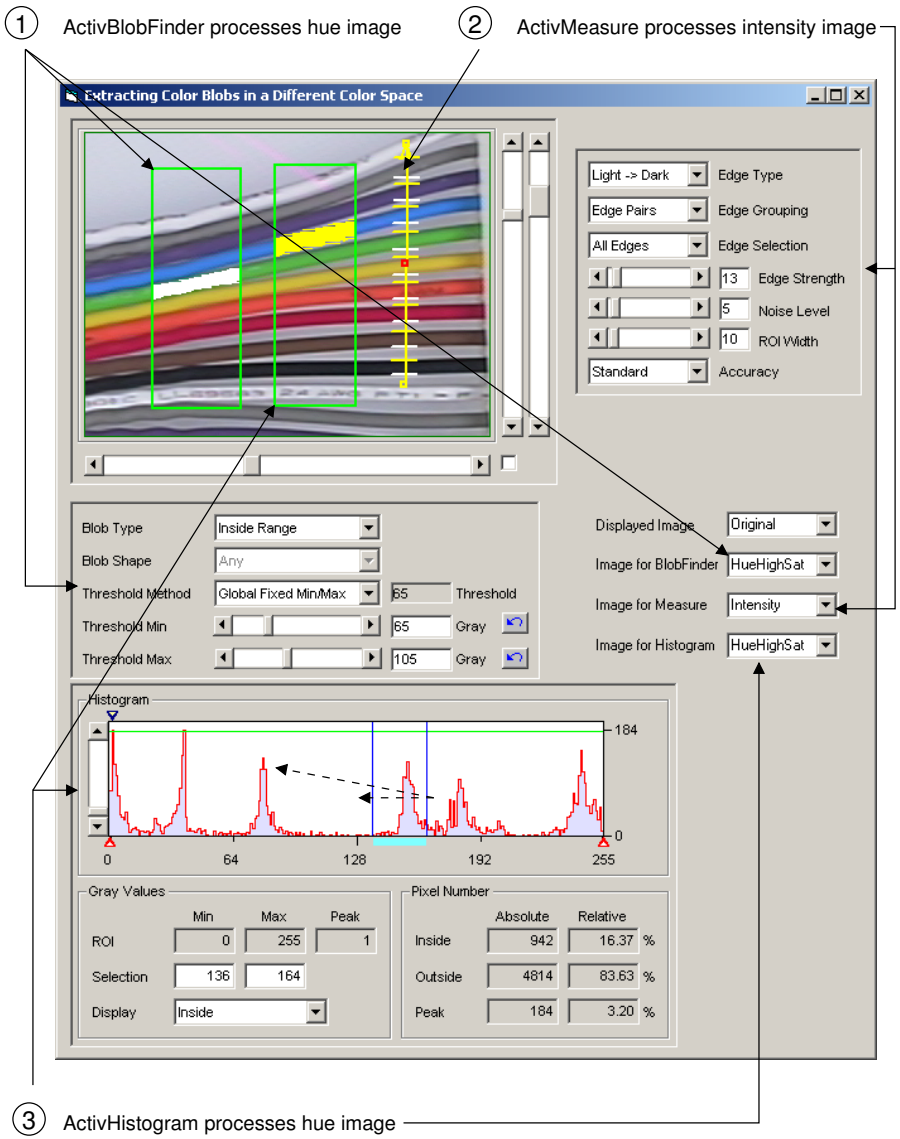


Figure 3.9: Processing color images in the color space HSI.

those parts that have high saturation values. The domain specifies which parts of the image are processed; if you create an ROI for an ActivVisionTool it is automatically combined with the

domain of the image, i.e., only those pixels are processed by the tool that are part of both the domain and of the ROI:

```
  Call hOpSet.SetSystem("store_empty_region", "true")
  Call hOpSet.Threshold(imageSaturation, imageHighSat, 30, 255)
  Call hOpSet.ReduceDomain(imageHue, imageHighSat, imageHueHighSat)

End Sub
```

In contrast to the previous example, the case that the input image is no color image is handled with the following code fragment, i.e., by aborting the execution cycle immediately using `AVTView`'s property `AbortCycle` and displaying a message in `AVTViewStatus` (or in a popup dialog if `AVTViewStatus` is not added to the form as in the example project):

```
If Not (AVTView1.HalconImageChannelCount = 3) Then
  AVTView1.AbortCycle = True
  Call AVTView1.SetStatusInfo("Not an RGB image!", eErrStatus)
  Exit Sub
End If
```

As depicted in figure 3.9 on page 81, you can select an image (channel) individually for each tool and for the display in a set of combo boxes. Similarly to the previous example, the tools check the current value of "their" box in the handler for the event `Start` and modify their input image accordingly, while the image to display is set in a handler for `AVTView`'s event `ImageGrabbed`.

The following code assures that the relevant parts of the application are reexecuted whenever you change the content of a combo box. For a tool, you can use a method of the tool's *registration*; to update the display you must reexecute all tools again, but of course without grabbing a new image:

```
Private Sub ComboDisplayedImage_Click()
  Call AVTView1.ExecuteAllToolsOnce(False)
End Sub
```

```
Private Sub ComboImageBlob_Click()
  Call AVTBlobFinder1.Registration.ToolsNeedRecalculation
End Sub
```

```
Private Sub ComboImageMeasure_Click()
  Call AVTMeasure1.Registration.ToolsNeedRecalculation
End Sub
```

```
Private Sub ComboImageHistogram_Click()
  Call AVTHistogram1.Registration.ToolsNeedRecalculation
End Sub
```

In section 4.4.1 on page 119, this example task is revisited and realized in a simplified way using ActivGenericInterface.

3.5 Handling DSC Files via the Programming Interface

As described in the Getting Started with ActivVisionTools, section 1.2.4 on page 9, ActivVisionTools automatically store the state of an application in the so-called DSC file whenever you start or stop the application. This automatic mode is useful when developing an application. However, there are cases where you might wish for more control on the saving and loading of the state. The ActivVisionTools programming interface helps you by providing the following methods and properties:

- The property `EnableWriteToDSCFile` lets you switch off the automatic saving. Typically, you do this in the final version of the application to prevent an accidental destruction of the DSC file.
- The method `WriteToDSC` saves the current state of the application in the DSC file at the moment you call the method. Thus, you can save an intermediate state, continue to configure the application, and return to the saved state - but only if you created a copy of the programmatically saved DSC file before it is overwritten by stopping the application.
- The method `ReadFromDSC` loads a specified DSC file. Thus, you can switch to another state at any moment and thereby, e.g., switch between different tasks.

Note, however, that the loading of DSC files is subject to the following limitations:

- The tools in the application loading the DSC file should be identical to the tools in the application that saved it.
- The method is rather slow (typically several seconds).
- To use different anchors or data code models, the configurations including the DSC files must be copied into separate directories. When using configurations in different directories, OCR fonts must either reside in the default OCR font directory, or be referenced using an absolute path, i.e., they should not reside in or under the project directory. It is also recommended to direct the results of ActivFile into such an absolute location.
- Certain non-persistent information is lost during reload (e.g., the active ROI, display selections, evaluation settings); furthermore, design-time properties are not affected.

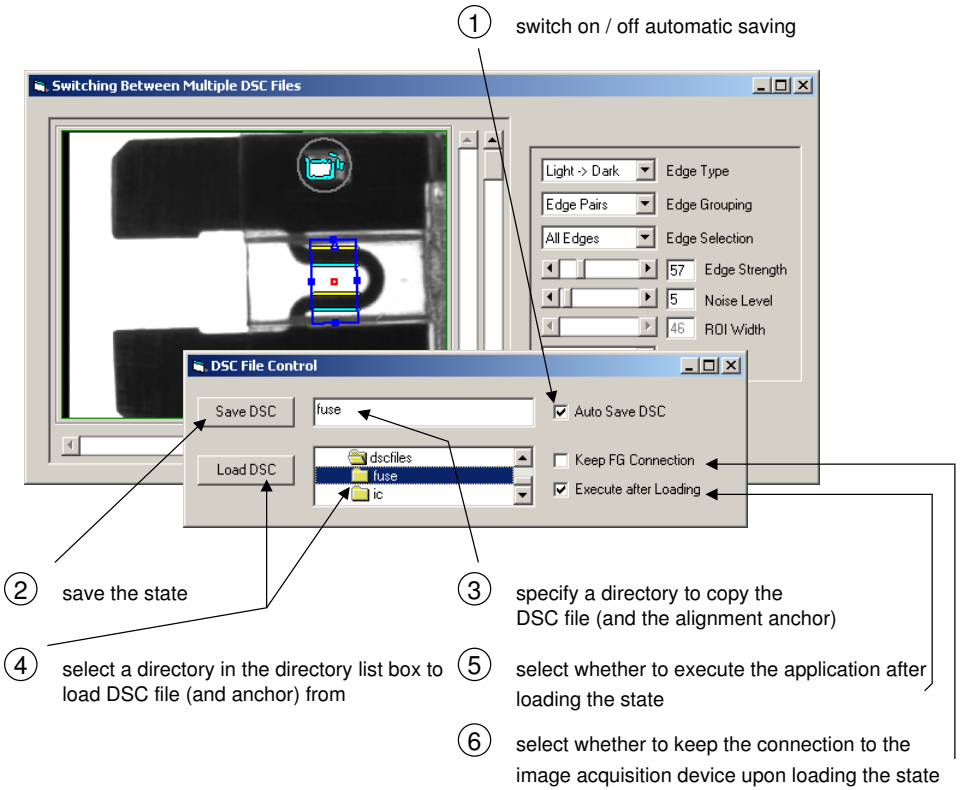


Figure 3.10: Interactive control of DSC files.

The example project `dscfiles\advanced_dscfiles.vbp` contains a graphical interface for the methods and properties above in form of a separate form that can be added to a project (see figure 3.10). The “main” form contains an aligned measuring application (see figure 3.10). To access the form providing DSC file control, the following lines of code are inserted; they open the other form and pass it the instance of `AVTView` and the name of the instance of `AVT-Alignment`:

```
Private Sub Form_Load()

  Call dsc_control.Show
  Call dsc_control.Initialize(AVTView1, AVTAlignment1.Name)

End Sub
```

The code described below stems from the second form. The method for initializing the form only requires the instance of `AVTView`; the instance names of `AVTAlignment` and `AVTDataCode` are optional. These tools need special treatment because here the state is not fully described in the DSC file, but also in the anchor and the data code model file, respectively.

```
Private atView As AVTView
Private sNameAlignment As String
Private sNameDataCode As String

Public Sub Initialize(atV As AVTView, _
                      Optional sAlignment As String = "", _
                      Optional sDataCode As String = "")

  Set atView = atV
  sNameAlignment = sAlignment
  sNameDataCode = sDataCode

End Sub
```

The property `EnableWriteToDSCFile` is linked to a check box (marked with ① in the figure):

```
Private Sub CheckAutoSaveDSC_Click()

  atView.EnableWriteToDSCFile = (CheckAutoSaveDSC.Value = vbChecked)

End Sub
```

The method `WriteToDSC` is called when the corresponding button (②) is pressed. To assure that the DSC file can indeed be written, `EnableWriteToDSCFile` is set to `'TRUE'` before calling `WriteToDSC` (and restored to its original value afterwards):

```
Private Sub SaveDSC_Click()

  Dim bWriteEnabled As Boolean
  bWriteEnabled = atView.EnableWriteToDSCFile
  atView.EnableWriteToDSCFile = True
  Call atView.WriteToDSC
  atView.EnableWriteToDSCFile = bWriteEnabled

End Sub
```

If you want to copy the DSC file into a separate directory, enter its name (relative to the current directory) in the text box (③); if the directory does not yet exist, it is created:

```
  Dim oFS As FileSystemObject
  Set oFS = New FileSystemObject

  Dim sBaseName As String
  sDirName = atView.NameDSCDir & TextSaveDSC.Text
  sBaseName = atView.NameDSCDir & atView.NameDSCFile

  If TextSaveDSC.Text = "" Then
    Exit Sub
  End If

  If Not oFS.FolderExists(sDirName) Then
    Call oFS.CreateFolder(sDirName)
  End If

  Call oFS.CopyFile(sBaseName & ".dsc", sDirName & "\")
```

The following code checks whether instance names of `AVTAlignment` or `AVTDataCode` have been passed; if yes, it copies the relevant files:

```
  If Not sNameAlignment = "" Then
    Call oFS.CopyFile(sBaseName & "_" & sNameAlignment & ".alm", _
                      sDirName & "\")
  End If
  If Not sNameDataCode = "" Then
    Call oFS.CopyFile(BaseName & "_" & sNameDataCode & ".dmd", _
                      sDirName & "\")
  End If
```

The method ReadFromDSC is called when you press the corresponding button (④); it looks for the DSC file in the directory selected in the directory list box:

```
Private Sub LoadDSC_Click()

  Dim sFileName As String

  sFileName = DirsDSCFiles.List(DirsDSCFiles.ListIndex) & _
              "\" & atView.NameDSCFile & ".dsc"
  Call atView.ReadFromDSC(sFileName, CheckKeepFG.Value)

End Sub
```

The remaining two check boxes control the behavior after loading a DSC file; you can select whether the application is executed after loading the new state (⑤) and whether the connection to the image acquisition device should be kept (⑥). The latter information is first used in the corresponding parameter of the method ReadFromDSC (see above); if the application is to be executed but the connection has not been kept, the connection is established by calling AVTView's method InitGrabber. By saving the current state of AVTView's property RunState before loading the DSC file and restoring it afterwards, you can even load DSC files while the application is running continuously:

```
  Dim bRunState As Boolean

  bRunState = atView.RunState

  If CheckAutoExec.Value Then
    If Not CheckKeepFG.Value Then
      Call atView.InitGrabber(False)
    End If
    If bRunState Then
      atView.RunState = bRunState
    Else
      Call atView.ExecuteAllToolsOnce
    End If
  End If
```

3.6 Using Multiple Instances of ActivView

Starting with ActivVisionTools 3.0, an application can contain multiple instances of ActivView. The main use of this is to acquire and process images from multiple cameras in a single application.

However, ActivView is a special ActivVisionTool: Besides its specific functionality (image acquisition, ROIs, display, etc.), it also provides the "operating system" of an application, i.e., it manages the connections between tools, executes them and collects their results. Therefore, using multiple instances of ActivView is not as straightforward as, e.g., using multiple instances of ActivMeasure.

Below, we explain the main principle of using multiple instances of ActivView; section 3.6.1 on page 90 describes the available modes, section 3.6.2 on page 103 contains additional information, e.g., how multiple instances of ActivView work together with alignment, calibration, or digital I/O.

In contrast to other ActivVisionTools, multiple instances of ActivView don't have the same functionality: One instance (the first you place on the form) is the so-called "master"; it provides both the specific functionality like image acquisition and the global one (connections between tools, execution, etc.). All other instances of `AVTView` are "slaves": They also acquire images, manage ROIs, and display results, but they are executed by the master `AVTView`, which also collects their results.

In fact, slave `AVTViews` are treated as *vision tools*. Therefore, the master executes them after *primary vision tools* like ActivAlignment (see figure 3.11). By default, slaves are executed after all "real" *vision tools* that are connected to the master `AVTView`, i.e., their property `Descriptor.Ordering` is set to a high value. As described in section 3.3.2 on page 63, you can modify this property via the programming interface.

Note that all slave `AVTViews` are connected to the master; you cannot connect them to other slaves.

! Via the check box ☑ `Execute Slaves` of `AVTViewExecute`, you can select whether slaves are executed (see figure 3.12 ① on page 90). Note that **by default slaves are not executed!** If you want to execute only some slaves and others not, you must specify the slave mode `'Independent'` for the latter (see section 3.6.1.4 on page 98).

You can check which instance of `AVTView` is the master at run time via `AVTViewConnections`. Figure 3.12 ② on page 90, e.g., depicts the tool connections corresponding to the project shown in figure 3.13 on page 93. If you click on a tool name in `AVTViewConnections`, the tool is highlighted on the form.

At design time, you can check who is master in the *Property Window* by looking up the property `IsMasterView` of the instances of `AVTView` (see figure 3.12 ② on page 90). This property can also be edited; however, this should not be necessary during normal development. One case

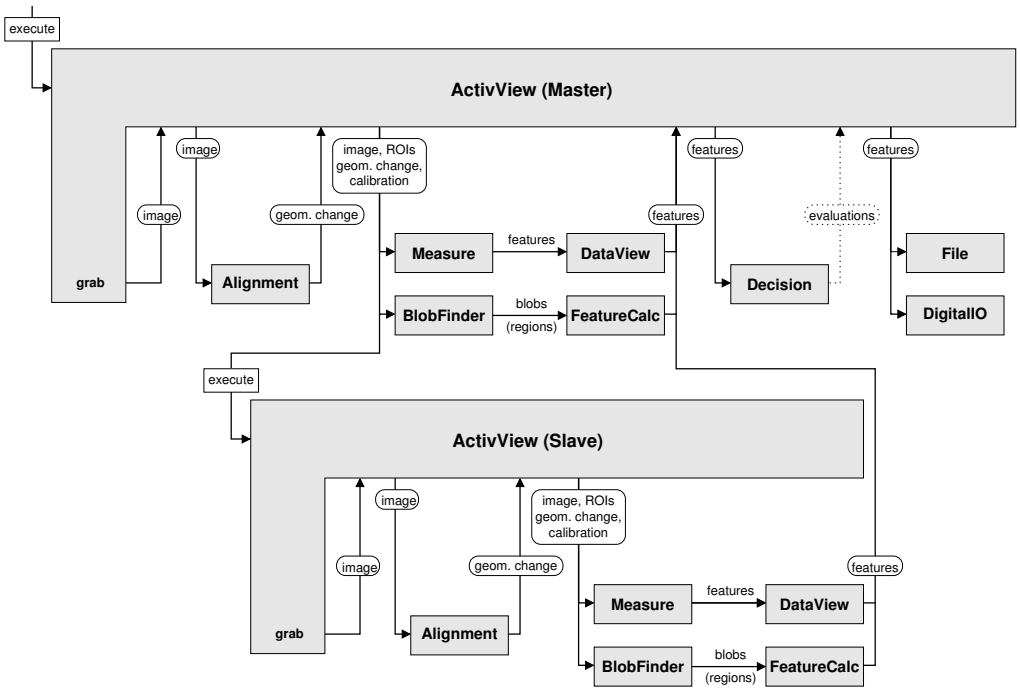


Figure 3.11: Input and output data flow with multiple instances of AVTView (slave mode: `'Grab Image On Execution'`).

where it becomes necessary is if you delete the master. Then, set `IsMasterView` to `'True'` for the instance that is to become the master. It may be necessary to close and open the form afterwards.

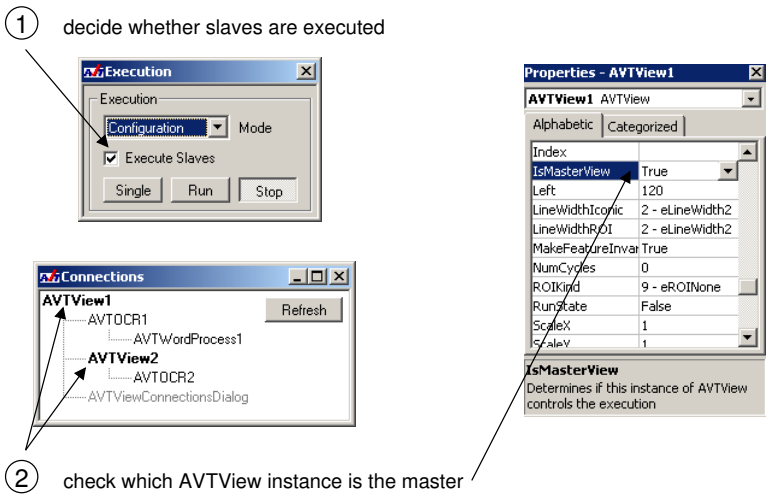


Figure 3.12: Basic settings of multiple AVTViews.

3.6.1 The Different Modes of Executing Slaves

As already mentioned, the main reason for using multiple instances of `AVTView` is to acquire and process images from multiple cameras. However, multiple `AVTViews` can also be useful in other scenarios, e.g., if you want to perform different machine vision tasks on the same image and display the results separately. Therefore, slave instances of `AVTView` can be executed in different modes.

This mode is stored in form of the property `SlaveMode`. It can be set at run time using `AVTViewProperties`, which can be opened by clicking on `AVTView` with the right mouse button and selecting `Properties` in the appearing context menu (see the User's Manual for ActivView, section 2.3 on page 20 for more information about this tool).

Below, the available slave modes and their typical application are listed. The following sections describe these modes with examples.

`'Grab Image On Execution'` (section 3.6.1.1 on page 92)
acquire and process images from multiple acquisition devices

`'Use Image From Predecessor'` (section 3.6.1.2 on page 94)
perform different tasks on the same image and display their results separately

`'Use Locally Stored Image'` (section 3.6.1.3 on page 96)
process multiple images that must be set via the programming interface, e.g., channels of

a color image, images acquired via *port switching* or simultaneously in form of a multi-channel image

`'Independent'` (section 3.6.1.4 on page 98)
scenarios in which tasks are not performed periodically in each execution cycle, e.g., if the result of one task determines which second task is executed or if images from a certain camera are only acquired and processed if a certain digital input signal is received

3.6.1.1 Acquiring and Processing Images From Multiple Cameras

The standard application of multiple instances of `AVTView` is to acquire and process images from multiple cameras. Therefore, the default value of `SlaveMode` is `'Grab Image On Execution'`. This means that to access multiple cameras from your ActivVisionTools application, in most cases all you need to do is to

1. place multiple instances of `AVTView` on the form at design time and execute the application,
2. configure the image acquisition for both instances via `AVTViewFG`, and
3. check ☑ `Execute Slaves` in `AVTViewExecute`.

The input and output data flow in this mode is depicted in figure 3.11 on page 89. As already noted, slave instances of `AVTView` (and the tools connected to them) are executed by the master `AVTView` by default after the *vision tools* that are directly connected to the master. The results of the slaves are collected together with those of the tools connected to the master, before they are all evaluated by ActivDecision or sent to an *output tool* like ActivFile or ActivDigitalIO. Slaves can use their own alignment and calibration data; examples for this can be found in section 3.6.2.2 on page 104 and section 3.6.2.3 on page 105, respectively.

The project `multiple_view_grabonexec\advanced_multiple_view_grabonexec.vbp` is a fully configured example for processing images from multiple sources. The task is to inspect packages using two cameras that observe different parts of the package. One reads the label, the other the last part of a type identifier (see figure 3.13). Note that you need a license for ActivOCR to run this example. Detailed information about ActivOCR can be found in the User's Manual for ActivOCR .

Note that this mode does not work for the following types of image acquisition devices: If your device accesses multiple cameras via *port switching*, please refer to section 5.2.1 on page 147 for more information; if it returns simultaneously grabbed images in form of a multi-channel image, please refer to section 5.2.2 on page 150. Detailed information about the interface to your image acquisition device can be found under `http://www.halcon.com/image-acquisition`.

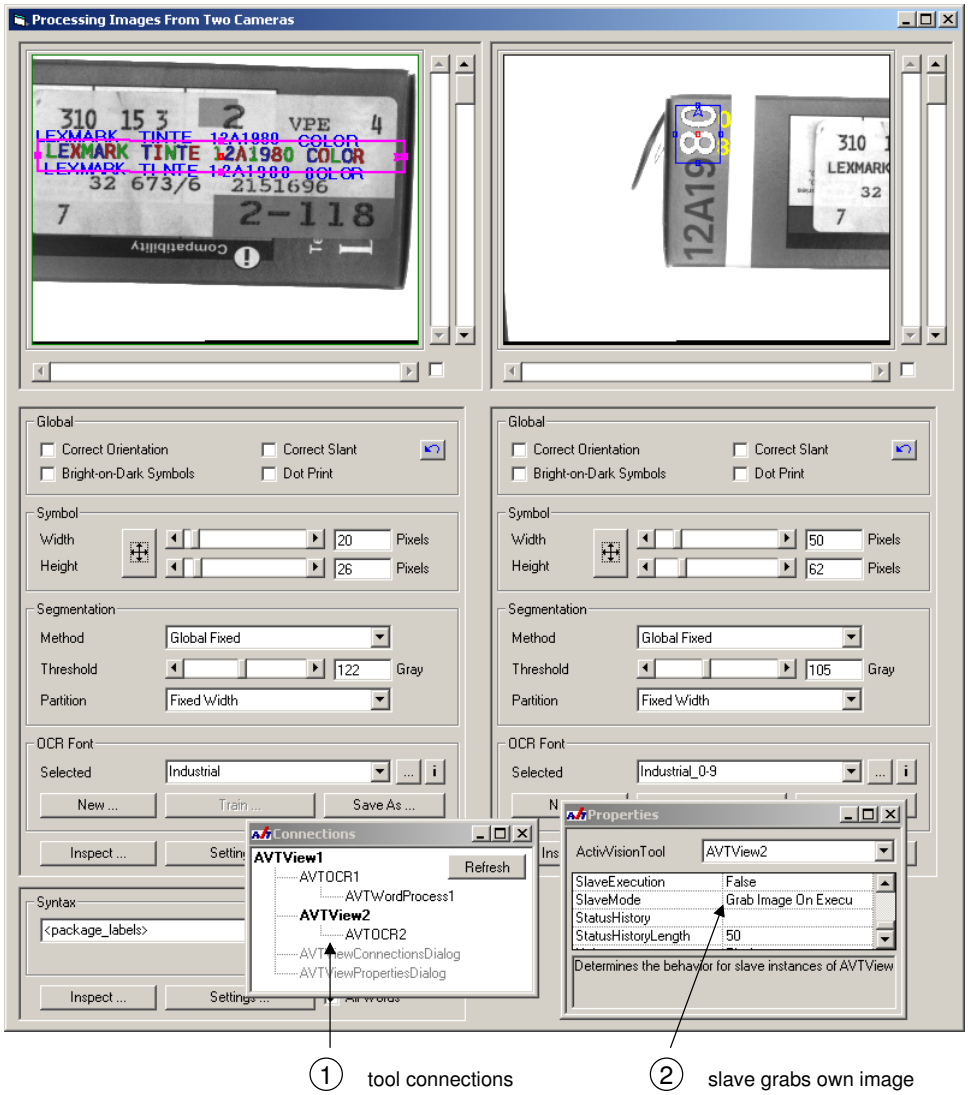


Figure 3.13: Inspecting a package using two cameras.

3.6.1.2 Performing Different Tasks on the Same Image

In the mode `'Use Image From Predecessor'`, slave `AVTViews` process the same image as the master. This mode is useful, e.g., if you want to display the results of different tasks separately.

The input and output data flow in this mode is depicted in figure 3.14. The main difference to the slave mode `'Grab Image On Execution'` (see figure 3.11 on page 89) is that the slave does not grab its own image but receives it from the master, together with alignment and calibration data. However, it can use its own instance of ActivAlignment and thus "overwrite" the master alignment.

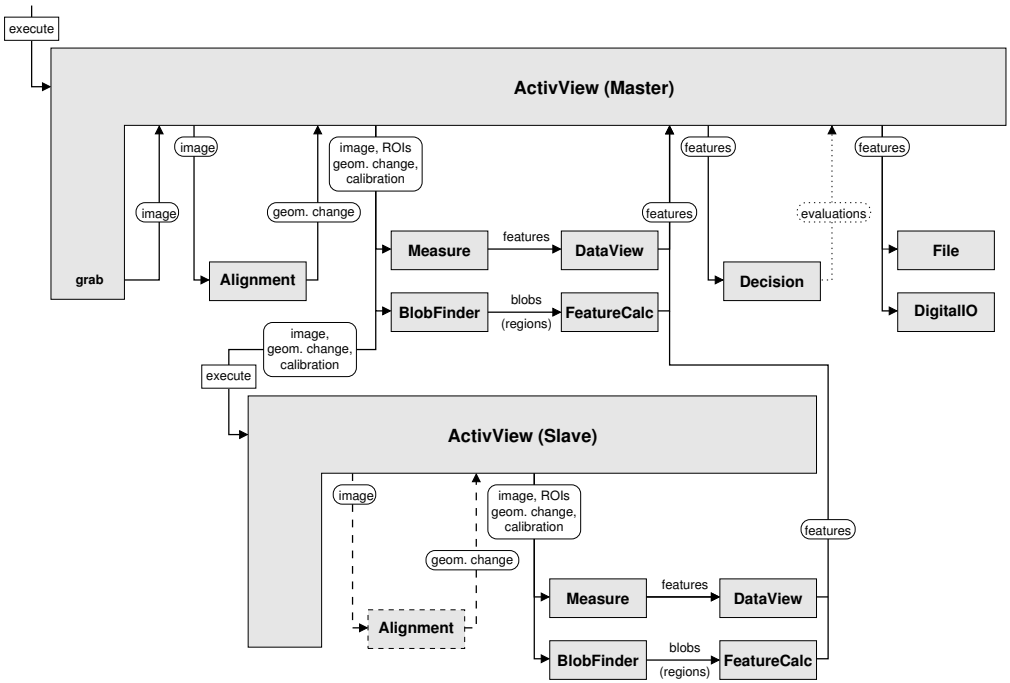


Figure 3.14: Input and output data flow for the slave mode `'Use Image From Predecessor'`.

`multiple_view_processinput\advanced_multiple_view_processinput.vbp` is a fully configured example project for this slave mode. It realizes the two-step blob extraction already described in section 3.4.1.2 on page 72 with two instances of `AVTView`, one for each step of the blob extraction. The code in the handler for the event `Finish` of the first instance of ActivBlobFinder remains almost the same; it uses the result of the first blob extraction (the white region) as the ROI for the second extraction step:

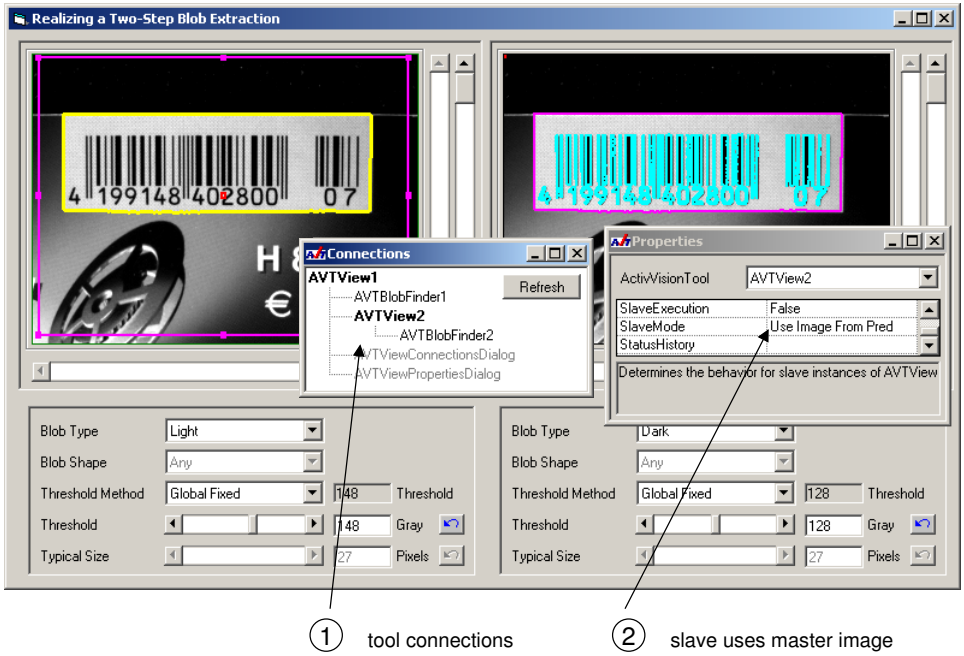


Figure 3.15: Example application performing two tasks on the same image: two-step blob extraction.

```
Private Sub AVTBlobFinder1_Finish(atIconicData As _
                                  ActivVTools.AVTImageMessageClass)

  Dim sROI1Name As String
  Dim hWhiteRegion As HUntypedObjectX

  sROI1Name = AVTBlobFinder1.Registration.ROIs(1).Name
  Set hWhiteRegion = atIconicData.IconicResults(sROI1Name)
  Call AVTView2.CreateROIRegion("AVTBlobFinder2", hWhiteRegion, "WhiteRegion")

End Sub
```

The main advantage of using multiple `AVTViews` is that the results of the two steps are now displayed in a clearer way (see figure 3.15). Furthermore, the execution sequence of the two instances of ActivBlobFinder doesn't need to be set via the programming interface, because the slave `AVTView` with its instance of ActivBlobFinder is automatically executed after the first ActivBlobFinder.

3.6.1.3 Using the Locally Stored Image

In the mode `'Use Locally Stored Image'`, slave `AVTViews` process the image that they find in their property `HalconImage` upon execution. This mode is useful if the image to be processed by a slave must be set via the programming interface of ActivVisionTools. Typical applications are if multi-channel images are split and then processed by different `AVTViews` (see below for an example that processes different channels of an RGB image and section 5.2 on page 147 for further examples).

The input and output data flow in this mode is depicted in figure 3.16. The main difference to the slave mode `'Grab Image On Execution'` (see figure 3.11 on page 89) is that the slave does not grab its own image but uses the locally stored one.

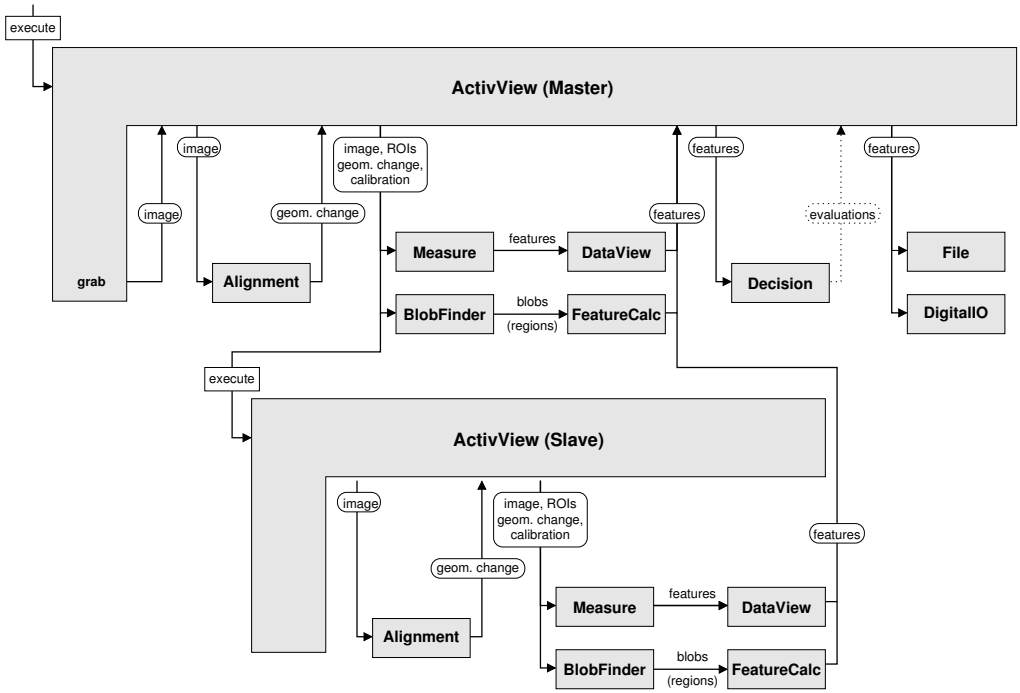


Figure 3.16: Input and output data flow for the slave mode `'Use Locally Stored Image'`.

`multiple_view_processcurrent\advanced_multiple_view_processcurrent.vbp` is a fully configured example project for this mode. It realizes the RGB color image inspection described in section 3.4.2.1 on page 76 with two instances of `AVTView`, which process the red and the blue channel, respectively (see figure 3.17). The code for splitting and distributing the image is now completely contained in the handler for the event `ImageGrabbed`:

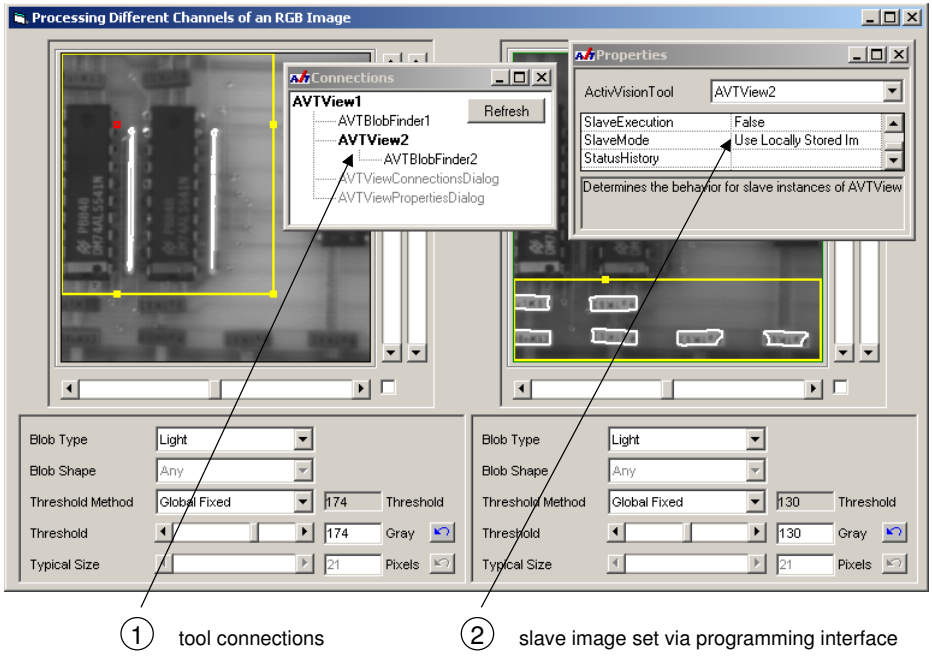


Figure 3.17: Example application processing multiple image channels.

```
Private Sub AVTView1_ImageGrabbed(hImage As HALCONXLib.HUntypedObjectX, _
                                  ByVal lError As Long)

  Dim imageRed As HUntypedObjectX, imageGreen As HUntypedObjectX
  Dim imageBlue As HUntypedObjectX

  If AVTView1.HalconImageChannelCount = 3 Then
    Set imageRed = AVTView1.HalconImageChannel(1)
    Set imageBlue = AVTView1.HalconImageChannel(3)
  Else
    Set imageRed = hImage
    Set imageBlue = hImage
  End If

  Set AVTView1.HalconImage = imageRed
  Set AVTView2.HalconImage = imageBlue

End Sub
```

3.6.1.4 Independent Execution

In the mode `'Independent'`, slave `AVTViews` are not executed by the master, but must be executed manually via the programming interface. This mode can be seen as a "escape hatch". It even allows to realize applications that don't fit the standard execution cycle of ActivVisionTools, e.g., if images from a certain camera are not acquired and processed in every execution cycle but only if a certain digital input signal is received. Another scenario is described below: Here, the result of a first inspection step is used to decide which second inspection task is performed.

The input and output data flow in this mode is depicted in figure 3.18. The main difference to the slave mode `'Grab Image On Execution'` (see figure 3.11 on page 89) is that the slave is not executed by the master. Note, however, that it still passes its results back to the master, which then evaluates them and sends them to the *output tools*.

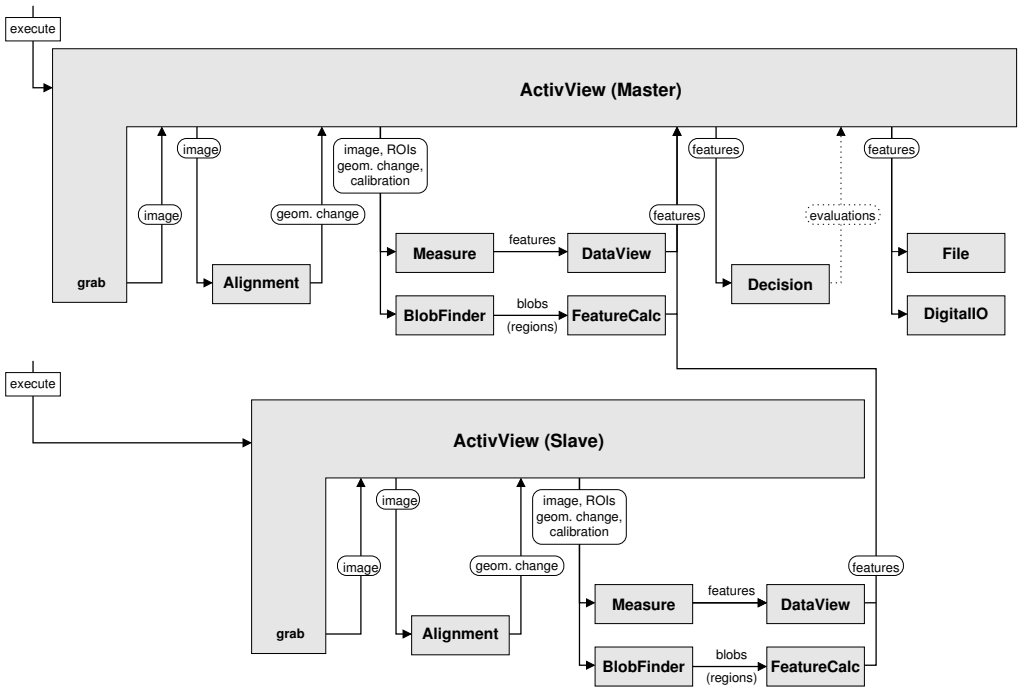


Figure 3.18: Input and output data flow for for the slave mode `'Independent'`.

`multiple_view_independent\advanced_multiple_view_independent.vbp` is a fully configured example project for this mode. It extends the package inspection application described in section 3.6.1.1 on page 92 in such a way that it can handle two types of packages (see

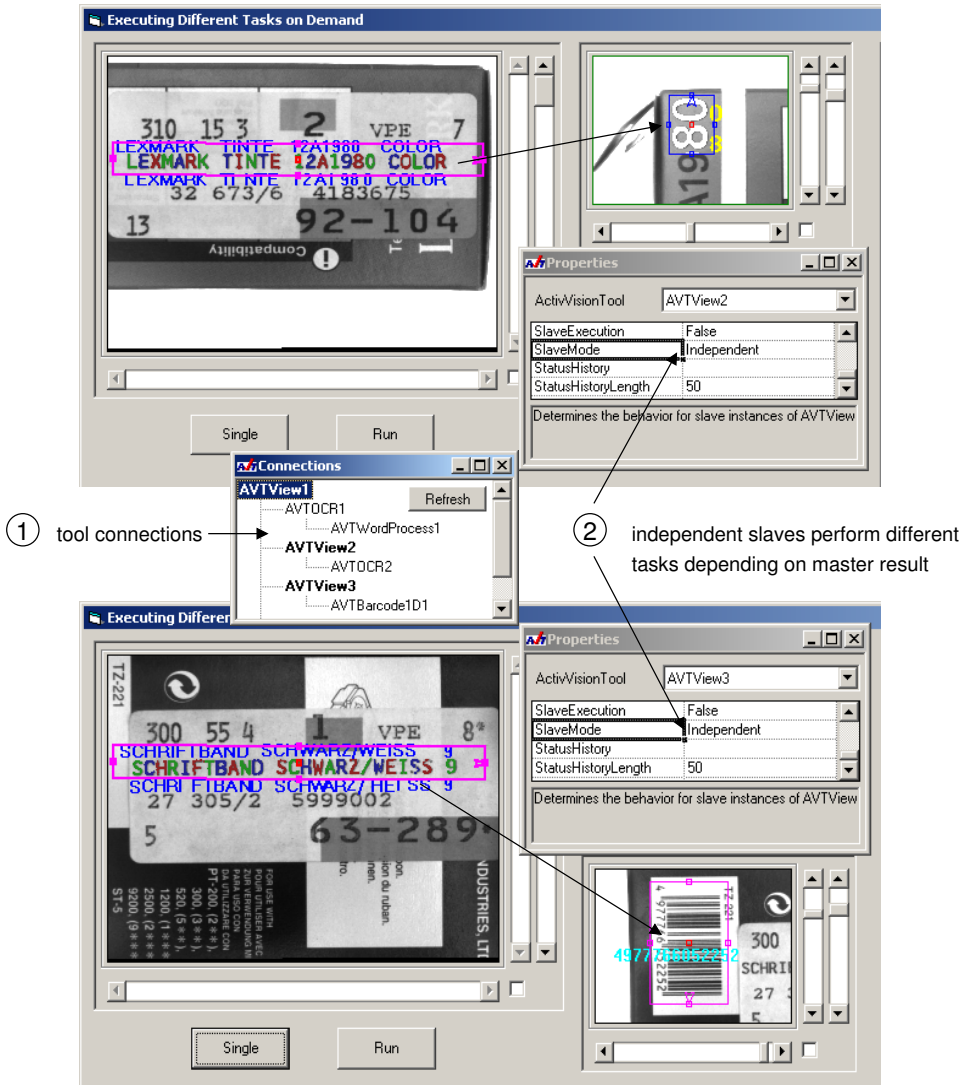


Figure 3.19: Example application with independently executed tasks.

figure 3.19): As before, in the first image the package label is read. Depending on the result, either the last part of a type identifier is read (upper part of the figure) or a bar code is searched and read (lower part of the figure). Note that you need a license for ActivOCR and ActivBarcode to run this example.

This switching between tasks can be accomplished in two ways, which correspond to the two main methods for using indepenent slaves:

1. Create your own execution cycle and execute master and slave `AVTViews` on demand.
2. Use the standard execution cycle but switch off tasks by setting their mode to `'Independent'`.

The example project contains code for both methods.

Creating a Customized Execution Cycle

In the example project, the new execution cycle is comprised in the following method, which first executes the master `AVTView` using `ExecuteAllToolsOnce` and then, based on its results, the appropriate slave:

```
Private Sub Inspect(bContinuousExec As Boolean)

  Call AVTView1.ExecuteAllToolsOnce(True, bContinuousExec)

  If sObjectType = "LEXMARK" Then
    Call AVTView2.ExecuteAllToolsOnce(True, bContinuousExec)
  End If
  If sObjectType = "SCHRIFTBAND" Then
    Call AVTView3.ExecuteAllToolsOnce(True, bContinuousExec)
  End If

End Sub
```

The selection of the appropriate slave is based on the first word read by the combination of ActivOCR and ActivWordProcess. This word is accessed in a handler for the event `Finish` of ActivWordProcess:

```
Private Sub AVTWordProcess1_Finish(atIconicData As _
                                   ActivVTools.AVTImageMessageClass, _
                                  atToolResults As Collection)

  Dim atToolResult As AVTToolResult
  Dim atWordProcessResult As AVTToolResult
  Dim bWordProcessResultsFound As Boolean
  Dim atROIResult As AVTROIResult

  bOCRResultsFound = False
  For Each atToolResult In atToolResults
    If atToolResult.Name = "AVTWordProcess1" Then
      Set atWordProcessResult = atToolResult
      bWordProcessResultsFound = True
    End If
  Next
  If bWordProcessResultsFound = False Then
    Exit Sub
  End If

  Set atROIResult = atWordProcessResult.ROIResults(0)
  If atROIResult.ObjectNum = 0 Then
    Exit Sub
  End If
  sObjectType = atROIResult.ObjectValue(handleWord, 0)

End Sub
```

The variables that are used in multiple methods are declared globally and initialized in the method `Form_Load`, which also initializes the image acquisition of the three instances of `AVTView`:

```
Private handleWord As Integer
Private sObjectType As String

Private Sub Form_Load()

  handleWord = AVTWordProcess1.FeatureHandleWord

  Call AVTView1.InitGrabber(False)
  Call AVTView2.InitGrabber(False)
  Call AVTView3.InitGrabber(False)

End Sub
```

The method realizing the execution cycle is called via the two buttons depicted in figure 3.19 on

page 99. One triggers a single cycle

```
Private Sub CommandSingle_Click()

  Call Inspect(False)

End Sub
```

while the other starts and stops a continuous execution with the help of a timer:

```
Private bIsRunning As Boolean

Private Sub CommandRun_Click()

  If bIsRunning Then
    TimerRun.Enabled = False
    CommandRun.Caption = "Run"
    bIsRunning = False
  Else
    bIsRunning = True
    CommandRun.Caption = "Stop"
    TimerRun.Enabled = True
  End If
End Sub
```

```
Private Sub TimerRun_Timer()

  Call Inspect(True)

End Sub
```

Using the Standard Execution Cycle

In fact, the new execution cycle realized above is not so different from the standard one: It just does not execute any slave `AVTViews`. This behavior can also be accomplished within the standard execution cycle by switching this slave's mode to `'Independent'` before it is executed and thereby "switching it off".

Note that this method is only possible if the information upon which the decision to switch off a slave is based is available before the slave is executed. If you take a look at the data flow depicted in figure 3.11 on page 89, you can see that this means that you can, e.g., switch off a slave in a handler to the event `Finish` of any of the previously executed *vision tools* but not in the event `Finish` of ActivDecision.

For our example project, this means that we can switch off the slaves in the event `Finish` of ActivWordProcess:

```
Private Sub AVTWordProcess1_Finish(atIconicData As _
                                    ActivVTools.AVTImageMessageClass, _
                                   atToolResults As Collection)

  ... access first word ...

    If sObjectType = "LEXMARK" Then
      AVTView2.SlaveMode = eSlaveGrabOnExecution
      AVTView3.SlaveMode = eSlaveIndependent
    End If
    If sObjectType = "SCHRIFTBAND" Then
      AVTView3.SlaveMode = eSlaveGrabOnExecution
      AVTView2.SlaveMode = eSlaveIndependent
    End If

End Sub
```

The example is configured in such a way that this method is used if you execute the application using `AVTViewExecute`.

3.6.2 Additional Information About Multiple ActivViews

3.6.2.1 Using Digital I/O

As described in the User's Manual for ActivDigitalIO, section 1.3.2 on page 8, ActivDigitalIO provides predefined input and output signals that are linked to the execution cycle. When you are using multiple `AVTViews`, this does not change. The only difference is that the following additional signals are provided for each slave:

- `'Allow Slave Grabbing'`
- `'Slave Grabbing Image'`
- `'Allow Slave Processing'`
- `'Slave Processing Image'`

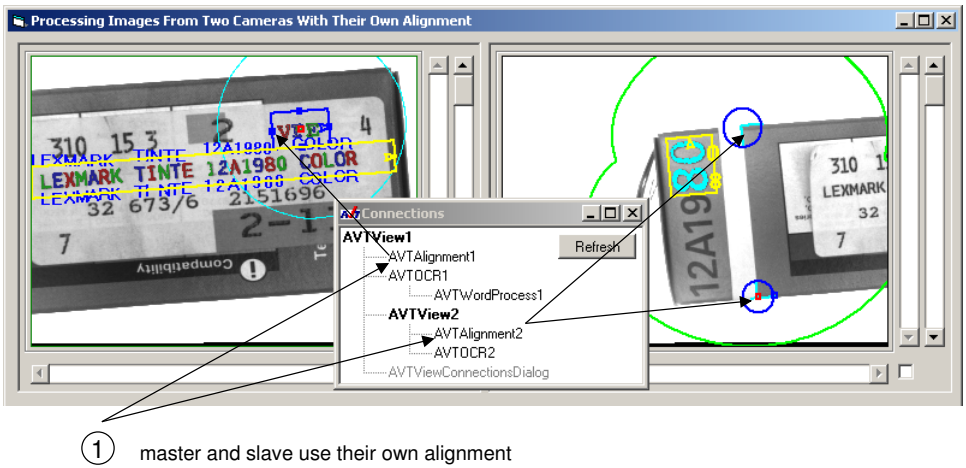


Figure 3.20: Master and slave use their own alignment.

3.6.2.2 Using Alignment

Using alignment with multiple instances of `AVTView` is straightforward: Each instance of `AVTView` can have its own instance of ActivAlignment (see the User's Manual for ActivAlignment for more information about this tool).

Figure 3.20 shows an extended version of the example project described in section 3.6.1.1 on page 92: Both instances of `AVTView` now have their own alignment; the master uses a part of the label as anchor, the slave two corners of a bright part on the package.

If you have a license for ActivAlignment, you can experiment with alignment by extending the example project `multiple_view_grabonexec\advanced_multiple_view_grabonexec.vbp`: Change the used image sequences to `package\ink_r2.seq` (master) and `package\ink_l2.seq` (slave). These sequences contain images of rotated packages that cannot be inspected without alignment. At design time, add two instances of `AVTAlignment` to the form and connect them to the two instances of `AVTView`. Train the anchors as described in the User's Manual for ActivAlignment. Now, the inspection should work even for the rotated packages.

As already mentioned in section 3.6.1.2 on page 94, slave `AVTViews` in the mode `'Use Image From Predecessor'` automatically inherit the alignment of the master, if they don't have their own instance of `AVTAlignment`. Note, however, that by default the alignment is switched off for the slave. To switch it on, set the property `AlignROIs` of the slave `AVTView` to `'True'` via `AVTViewProperties`, which can be opened by clicking on `AVTView` with the

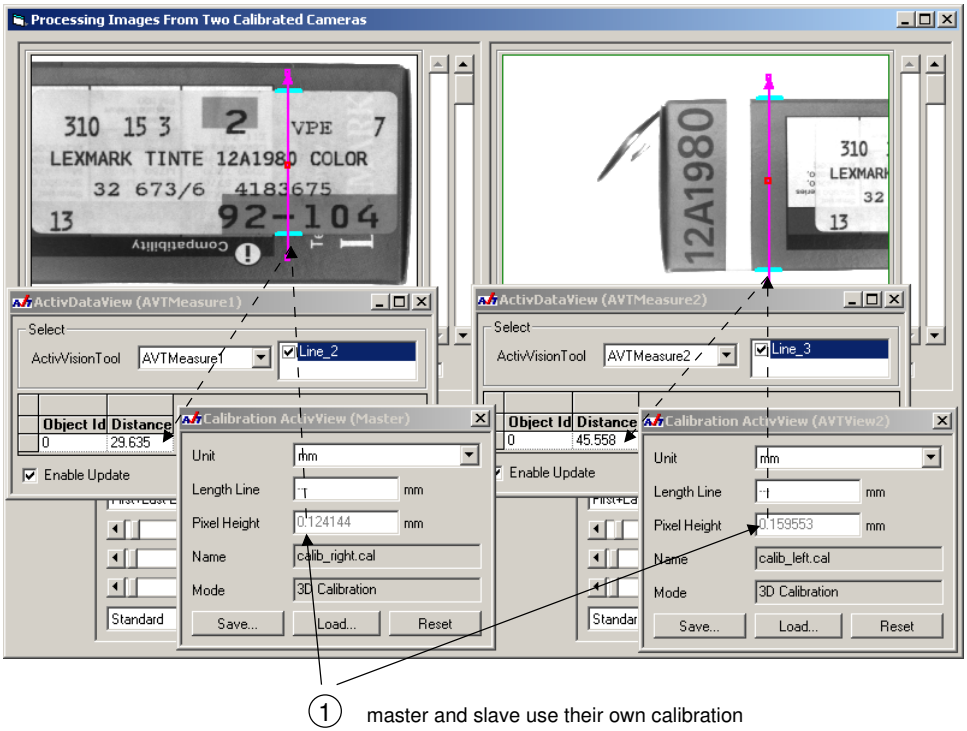


Figure 3.21: Master and slave use their own calibration.

right mouse button and selecting `Properties` in the appearing context menu (see the User's Manual for ActivView, section 2.3 on page 20 for more information about this tool).

3.6.2.3 Using Calibration

Analogously to alignment, each instance of `AVTView` can use its own calibration data, which can stem from `AVTViewCalibration` (see e.g. the User's Manual for ActivView, section 2.5 on page 24) or ActivGeoCalib (see the User's Manual for ActivGeoCalib). In figure 3.21, e.g., both cameras have been calibrated using ActivGeoCalib, allowing high-precision measurements in metric coordinates.

As with the alignment, slave `AVTViews` in the mode `'Use Image From Predecessor'` automatically inherit the calibration data of the master, which is of course what you expect because they use the same image.

Chapter 4

Creating Own ActivVisionTools

This chapter shows how to use ActivGenericInterface to integrate additional functionality, e.g., image (pre)processing steps, into an ActivVisionTools application. Please note that most of the corresponding Visual Basic example projects use HALCON, thus in order to run them, you need license for the (virtual) tool ActivHALCONFoundation (or a HALCON license).

4.1 Introducing ActivGenericInterface 108
4.2 Some Words About HALCON 108
4.3 How to Create a New ActivVisionTool 110
4.3.1 Basic Configuration 110
4.3.2 Accessing the Input Data 112
4.3.3 Performing the Actual Processing 113
4.3.4 Setting the Output Data 116
4.3.5 Displaying Results 116
4.4 Examples 118
4.4.1 Image Preprocessing Tools for HSI Color Processing 119
4.4.2 A Chain of Tools to Extract and Analyze Contours 121
4.4.3 Using Calibration and Alignment Information 137

4.1 Introducing ActivGenericInterface

At first glance, ActivGenericInterface is a remarkably inactive ActivVisionTool. In fact, one might say that it is just an empty "shell" that waits to be filled with additional functionality you want to integrate into an ActivVisionTools application. However, ActivGenericInterface is more than a container: Like the other ActivVisionTools, it is equipped with internal mechanisms that allow to integrate it with a few mouse clicks into an ActivVisionTools application, i.e., it automatically receives its required input data and passes its results to other tools without any action required from your part. Furthermore, ActivGenericInterface provides methods that allow to access input data and set output data easily, without having to worry about low-level data structures.

But nevertheless, an instance of ActivGenericInterface starts out as an empty container, which you can then mould into any form you like by choosing its tool type (*vision tool*, *result processing tool*, *output tool*, etc.) and which type of input and output data (image, regions, features, etc.) it requires and produces, respectively. Thus, you may create a variety of new ActivVisionTools, e.g., for preprocessing the grabbed images, for performing more complex image processing steps, or for communicating with special hardware devices. Section 4.3 on page 110 shows how to turn an instance of ActivGenericInterface into a new ActivVisionTool, with examples following in section 4.4 on page 118.

Please note that you cannot generate a user control from an ActivGenericInterface. This implies that in order to reuse an ActivGenericInterface, you must copy the corresponding code.

A final remark regards the appearance of ActivGenericInterface: This tool has no graphical user interface of its own. Thus, it is invisible at run time; at design time, it is represented by its icon .

4.2 Some Words About HALCON

The main use of ActivGenericInterface is to integrate additional image processing functionality into an ActivVisionTools application. To realize the image processing itself, you need HALCON, MVTec's unique machine vision library, because this is the base of all image processing in ActivVisionTools.

If you are already using HALCON on its own, i.e., if you have a license for it, you can immediately start to use it within your ActivVisionTools application. If not, you can purchase ActivHALCONFoundation, which allows to use HALCON in conjunction with ActivVisionTools.

To be more exact, ActivHALCONFoundation allows you to use HALCON's module 'Foundation', which contains more than 80% of HALCON operators, not only basic operators (e.g., I/O, visualization) but also fast and powerful image processing methods like blob and texture

analysis, edge and point detectors, color image processing, geometric transformations, FFT and much more. However, it does not include advanced functionality like subpixel-precise edge detection, data code reading, or stereo image processing. If you need this functionality within an ActivVisionTools application, please contact your local distributor.

The following parts of HALCON's documentation are useful in combination with ActivVisionTools:

- The manual **HALCON Quick Guide** introduces HALCON in general.
- The **HALCON Programmer's Guide** shows how to use the HALCON library in different programming languages, e.g., Visual Basic or C++.
- The **HALCON Reference Manual** describes the HALCON operators in detail. It is available in different versions, corresponding to HALCON's programming language interfaces. When programming in Visual Basic, Visual Basic .NET, or C#, you typically use the HALCON/COM interface, when programming in Visual C++ the HALCON/C++ interface.

If you are not already using HALCON, you can either download these manuals from MVTec's WWW server (`http://www.halcon.com/documentation`) or install them from your ActivVisionTools CD as follows (administrator privileges necessary): Insert the CD and, if necessary, cancel the automatically starting setup of ActivVisionTools. Then execute HALCON's setup program `Setup.exe` located in the directory `halcon/nt-x86` and let it install the documentation. This method has the advantage that the documentation will automatically appear in the Windows start menu.

4.3 How to Create a New ActivVisionTool

With the following five steps, you turn an instance of ActivGenericInterface into a new ActivVisionTool:

1. Basic configuration
2. Accessing the input data
3. Performing the actual processing
4. Setting the output data
5. Displaying results

In the following sections, these steps are described in detail; example applications follow in section 4.4 on page 118.

4.3.1 Basic Configuration

! In order to use the programming interface of ActivGenericInterface within Visual Basic, you must **add the ActivVisionTools type library and the HALCON/COM library to the project's references** by checking the corresponding boxes in the Visual Basic menu dialog `Project` ▷ `References`. Note that you can add the HALCON/COM library without having a HALCON license; this license is only required if you want to use HALCON operators.

Name

After you create a new instance of `AVTGenericInterface` by double-clicking in the toolbar, it has a default name, e.g., `AVTGenericInterface1`. Especially if your application contains more than one instance of `AVTGenericInterface`, you might want to change their names. For ActivVisionTools in general, this is not recommended, because it causes problems for the internal mechanisms governing the connections of the tools.

For `AVTGenericInterface`, however, there is a safe way to rename it:

1. Change the name **immediately** after placing the tool on the form.
2. Then close the form and open it again

! **Never rename a tool if it has already been executed!**

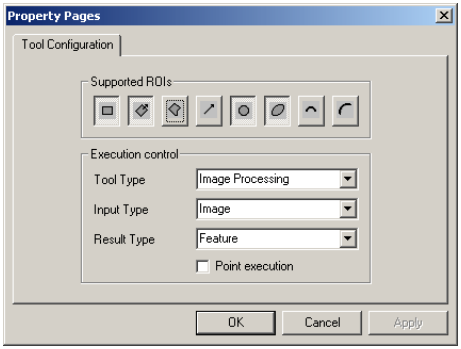


Figure 4.1: An example tool configuration.

Tool Type, Input and Output Data

You can specify the general behavior of your new tool at design time via the property page `Tool Configuration` of `AVTGenericInterface` (see figure 4.1 for an example). To open the property page, click on the tool with the right mouse button and select `Properties` in the appearing context menu.

1. In the frame `Supported ROIs`, select those ROI shapes the tool supported by clicking the corresponding icon. If the tool does not have ROIs, we recommend to unselect all shapes.
2. Select the desired tool type in a corresponding combo box (see section 3.1 on page 54 for more information about tool types).
3. Below, specify which type of input data the tool needs and which type of output data it creates. reftableexecution:types lists the possible data for the different tool types.
4. If the tool is to act as a so-called *pointTool*, i.e., if it is to be triggered if you move the mouse over `AVTView`, check the box ☑ `Point execution`.

Connections

By default, your new tool is connected to the first instance of `AVTView`. To change this, open `AVTViewConnections` (also see the User's Manual for ActivView, section 3.5 on page 36) by clicking on `AVTView` with the right mouse button and selecting `Connections` in the appearing context menu.

Initializing the Features

If the new tool is to calculate features, i.e., if you selected `'Feature'` in the combo box `Result Type`, you must specify these features before the tool is executed. The place to perform this specification is in a handler for the event `InitializeFeatures`, which is raised by an `AVTGenericInterface` when the application is started. In the handler, you first

- specify the total number of calculated features using `AVTGenericInterface`'s method `SetFeatureNum` and then
- describe the individual features using the method `AddFeature`, specifying details like its name, data type and unit and whether the feature belongs to the tool, an ROI, or an object (see the **Reference Manual** for more information).

The following example code creates three features for an `AVTGenericInterface` named `AVTMyTool`: a tool feature containing a string, an ROI feature containing an integer value without a unit, and an object feature containing floating point values, which can have a metric unit, i.e., which correspond to a length.

```
Private handleTFeat As Integer
Private handleRFeat As Integer
Private handleOFeat As Integer

Private Sub AVTMyTool_InitializeFeatures()
  Call AVTMyTool.SetFeatureNum(3)
  handleTFeat = AVTMyTool.AddFeature("A Tool Feature", eFeatTool, _
                                     eTypeWChar, "AA", "ZZ", eUnitDimNone, _
                                     eEvalIntString, 1, 20)
  handleRFeat = AVTMyTool.AddFeature("An ROI Feature", eFeatROI, _
                                     eTypeInt, 0, 9999, eUnitDimNone, _
                                     eEvalIntNumber, 1, 1)
  handleOFeat = AVTMyTool.AddFeature("An Object Feature", eFeatObject, _
                                     eTypeDouble, -10#, 10#, eUnitDimMeter)
End Sub
```

4.3.2 Accessing the Input Data

In contrast to the other ActivVisionTools, `AVTGenericInterface` does not raise the events `Start` and `Finish` (compare section 3.1 on page 54). Instead, it only raises the event `Execute` (or `ExecutePoint` if it is a *point tool*). Therefore, you must place the main code for your new ActivVisionTool in a handler for this event.

The first step is to access the input data provided by the predecessor. To facilitate this, `AVTGenericInterface` provides a set of methods and properties:

- **Image:** `HalconImage`, `GetReducedImage`
  The ID of the image to be processed can be queried (and modified) via `AVTGenericInterface`'s property `HalconImage` (see section 4.4.1 on page 119 for an example).

  In many applications, not the complete image is to be processed, but only the part(s) inside a (set of) ROI (s). By calling the method `GetReducedImage` with an ROI as parameter, you can "reduce" the image to the specified ROI (see section 4.4.2 on page 121 for an example).

- **ROIs:** `ROIs`, `GetPredecessorResultsIconicROIs`
  If the tool has own ROIs, the property `ROIs` contains them in form of a `Collection` of `ActivVTools.AVTROIClass`.

  If the tool uses the ROIs of its predecessor (as ActivFeatureCalc uses those of ActivBlobFinder), the corresponding `Collection` can be queried with the method `GetPredecessorResultsIconicROIs`.

  Examples for both methods can be found in section 4.4.2 on page 121.

- **Iconic results:** `GetPredecessorResultsIconicData`
  You can access the iconic results of the predecessor using the method `GetPredecessorResultsIconicData`. As a parameter, you must specify an ROI, which of course must be part of the `Collection` returned by the method `GetPredecessorResultsIconicROIs` (see above). `GetPredecessorResultsIconicData` returns the iconic results in form of an `HALCONXLib.HUntypedObjectX`; to further process the results, you need HALCON. See section 4.4.2 on page 121 for examples.

- **Feature results:** `GetPredecessorResultsFeatureData`
  The features calculated by the predecessor(s) and the corresponding evaluations are returned by the method `GetPredecessorResultsFeatureData` in form of a `Collection` of `ACTIVVTOOLSLib.AVTToolResult` (see section 4.4.2 on page 121 for an example; more information about this class can be found in the manuals of those ActivVisionTools that calculate features, i.e., ActivMeasure, ActivBarcode, ActivFeatureCalc, and ActivAlignment).

- **Calibration and alignment data:** `Calibration`, `AffineTransformation`
  The calibration and alignment data is stored in the properties `Calibration` and `AffineTransformation`, respectively (see section 4.4.3 on page 137 for examples).

4.3.3 Performing the Actual Processing

Of course, we cannot relieve you of the task of creating the code performing the actual processing. In this section, we therefore just point out some useful programming strategies; for Visual Basic examples, please refer to section 4.4 on page 118.

If your new ActivVisionTool is to process iconic data, i.e., the image or iconic results of a pre-

decessor, we recommend to perform the processing inside a loop over all ROIs. The following code, e.g., can serve as a skeleton for a tool that processes the image inside a set of ROIs:

```
Private Sub AVTGenericInterface1_Execute()

  Dim atROI As AVTROIClass
  Dim hImageInROI As HUntypedObjectX

  On Error GoTo ErrorHandler

  If AVTGenericInterface1.HalconImage Is Nothing Then
    Exit Sub
  End If

  For Each atROI In AVTGenericInterface1.ROIs
    Set hImageInROI = AVTGenericInterface1.GetReducedImage(atROI)
    ' process image part in ROI
  Next

  Exit Sub
ErrorHandler:
  Debug.Print Err.Description
End Sub
```

Besides the processing loop, the code shown above contains additional useful parts: First of all, we recommend to **always supply an error handler** when creating a handler for `AVTGeneric-Interface`'s event `Execute` (or `ExecutePoint`). Secondly, it is useful to check whether `AVT-GenericInterface` is executed with valid input data, and to exit the handler if this is not the case. The code shown above, e.g., explicitly checks whether it received a valid image and, by using the `For Each...Next` loop, implicitely checks whether ROIs were created for the tool.

The following code example could serve as a skeleton for a tool that processes the iconic results of a predecessor; its core is a loop over all the ROIs of the predecessor that produced iconic results:

```
Private Sub AVTGenericInterface2_Execute()

  Dim atROI As AVTROIClass
  Dim hIconicResult As HUntypedObjectX

  On Error GoTo ErrorHandler

  For Each atROI In AVTGenericInterface2.GetPredecessorResultsIconicROIs
    Set hIconicResult = _
      AVTGenericInterface2.GetPredecessorResultsIconicData(atROI)
    ' process iconic result
  Next

  Exit Sub
ErrorHandler:
  Debug.Print Err.Description
End Sub
```

In contrast, if a tool is to process feature results, e.g., to output them via a special hardware device, a suitable approach is to traverse the tree of results as in the following code example (error handler left out):

```
Private Sub AVTGenericInterface3_Execute()

  Dim atToolResult As AVTToolResult
  Dim atROIResult As AVTROIResult
  Dim iNumROIs As Integer, iNumObjects As Integer
  Dim i As Integer, j As Integer
  Dim vValues As Variant

  For Each atToolResult In _
                        AVTGenericInterface3.GetPredecessorResultsFeatureData
    iNumROIs = atToolResult.ROINum
    For i = 0 To iNumROIs - 1
      Set atROIResult = atToolResult.ROIResults(i)
      iNumObjects = atROIResult.ObjectNum
      vValues = atROIResult.ObjectValues(handleFeat)
      For j = 0 To iNumObjects - 1
        ' process feature value
      Next
    Next
  Next

End Sub
```

For more information about the tree of results please refer to the manuals of those ActivVision-

Tools that calculate features, e.g., ActivMeasure (see section 4.3.1 on page 31), ActivBarcode (see section 4.3.1 on page 27), ActivFeatureCalc (see section 4.2.1 on page 35), ActivOCR (see section 4.3.1 on page 63), or ActivAlignment (see section 4.3 on page 28), .

4.3.4 Setting the Output Data

`AVTGenericInterface` provides the following methods for setting output data:

- **Image:** `HalconImage`
  As already described in section 4.3.2 on page 112, `AVTGenericInterface`'s property `HalconImage` stores the ID of the processed image. If you modify this property, i.e., store the ID of a modified image, direct successors of the tool process this image instead of the original one (see section 4.4.1 on page 119 for an example). Please refer to section 3.4.2 on page 73 for more information about modifying images and its effect on other tools.
- **Iconic results:** `SetIconicResult`
  With the method `SetIconicResult`, you set the iconic results in form of an `HALCONXLib.HUntypedObjectX` for a specified ROI (see section 4.4.2 on page 121 for examples).
- **Feature results:** `SetFeatureResultsObjectNum`, `SetFeatureResultsObject`, `SetFeatureResultROI`, `SetFeatureResultTool`

  ! If the tool calculates object features, you must **first specify the number of objects** using the method `SetFeatureResultsObjectNum`, because it resets the data structure containing the results, including the tool and ROI features! Then, call the method `SetFeatureResultsObject` to set the object features. Note that with this method you set the values of all objects for one specified feature. Tool and ROI features can be set via the methods `SetFeatureResultROI` and `SetFeatureResultTool`, respectively.

4.3.5 Displaying Results

`AVTGenericInterface` provides methods to display the (iconic) results of your new ActivVisionTool in `AVTView`, overlaid on the image. In fact, there are three "sources" of display, which can be assigned different colors in `AVTViewDisplayModes`: A tool can display its results, e.g., ActivMeasure the extracted edges or ActivBlobFinder the extracted blob regions; these results can be assigned to two different groups, each of which is displayed in its own color. Secondly, ActivDecision marks objects evaluated as "not okay" in a separately choosable color (typically in red). Finally, if you click on an object in ActivDataView or ActivDecision's result tree, it is highlighted in another color, which can also be selected in `AVTViewDisplayModes`.

In `AVTGenericInterface`, these three display mechanisms are covered by two methods:

1. To display the (iconic) results, call the method `AddOverlay`, passing the results in form of an `HALCONXLib.HUntypedObjectX` and the ROI that produced these results as parameters.
2. The method `SetFeatureObjects` covers the other two display mechanisms, i.e., the display of error results and the highlighting when selecting an object in ActivDataView or ActivDecision. As parameters, you must pass an ROI and its objects, as usual in form of an `HALCONXLib.HUntypedObjectX`.

Typically, you use `AddOverlay` for tools that produce iconic results, and `SetFeatureObjects` for tools that calculate features; see section 4.4.2 on page 121 for examples. Note that both methods check internally whether results are to be displayed at all, which can be configured in `AVTViewConfigExec`.

`AVTGenericInterface` provides yet another display method: `AddOverlayText` lets you display text in `AVTView`. An example can be found in section 4.4.3 on page 137. Furthermore, you can display arbitrary circles and lines with the methods `AddOverlayCircle` and `AddOverlayLine`.

4.4 Examples

In the following, we apply the methods described in section 4.3 on page 110 in four example Visual Basic applications. The corresponding example projects are already fully configured:

- `color\advanced_color_generic.vbp` (section 4.4.1)
  This example revisits the task of color image processing, which was presented in section 3.4.2.2 on page 80. It solves the task by creating two new *vision tools*, which "preprocess" the original image, i.e., select the color channel which is suited best for the needs of their successor tools. Please note that you need a license for ActivHALCONFoundation (or HALCON) to run this example!

- `contours\advanced_contours.vbp` (section 4.4.2 on page 121)
  This example creates a chain of new tools with the goal of extracting and analyzing contours: Head of the chain is a *vision tool* that extracts contours and outputs them as its iconic results. It is followed by a second *vision tool*, which splits the contours into segments, narrows them down to full circles and straight line segments, and passes them on together with features indicating their type and the overall number of circles and lines, respectively. Finally, two *result processing tools* follow that work in parallel: One analyzes the straight line segments, the other one the circles. The contour segments are represented by *XLDs* (eXtended Line Descriptions). Of course, the whole functionality could also be placed into a single tool.

  Please note that you need a license for ActivHALCONFoundation (or HALCON) to run this example!

- `calibration\advanced_calibration.vbp` (section 4.4.3.1 on page 137)
  This example shows how to convert numeric data into world coordinates. It creates an *inspection tool*, which is also a *point tool*; similarly to ActivZoom, the current position of the mouse in the image is converted, but now displayed in `AVTView`.

- `alignment\advanced_alignment.vbp` (section 4.4.3.2 on page 141)
  This example shows how to use alignment information in your new ActivVisionTool. Similarly to the example described above, a new *inspection tool* displays the coordinates of the current position of the mouse in the image, but now relative to the coordinate system of the reference image.

4.4.1 Image Preprocessing Tools for HSI Color Processing

The Visual Basic example `color\advanced_color_generic.vbp` revisits the task of color image processing, which was presented in section 3.4.2.2 on page 80. Instead of letting you select the image (channels) for the tools to process via combo boxes, two instance of `AVTGenericInterface` are now in charge of supplying suitable images to the tools: One instance is to pass the hue image to ActivBlobFinder and ActivHistogram, the other one passes the intensity image to ActivMeasure, as depicted in figure 4.2.

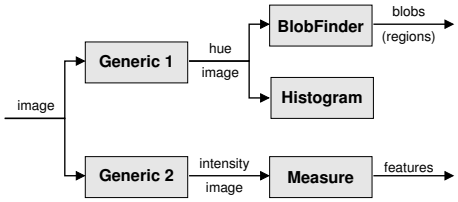


Figure 4.2: Tool connections in the example for color image processing.

To realize this, two instances of `AVTGenericInterface` are created, renamed to `AVTHueImage` and `AVTIntensityImage`, and configured as shown in figure 4.3: Both are declared as *vision tools* that require images as input and output images as well; therefore, ActivBlobFinder, ActivMeasure, and ActivHistogram can be connected to them, as they require images as input. In other words, the two new tools act as image preprocessors.

The code in the handlers for the event `Execute` of the two tools is very simple; the tools just set their output image to the hue and intensity channel, respectively:

```
Private Sub AVTHueImage_Execute()
  Set AVTHueImage.HalconImage = imageHueHighSat
End Sub

Private Sub AVTIntensityImage_Execute()
  Set AVTIntensityImage.HalconImage = imageIntensity
End Sub
```

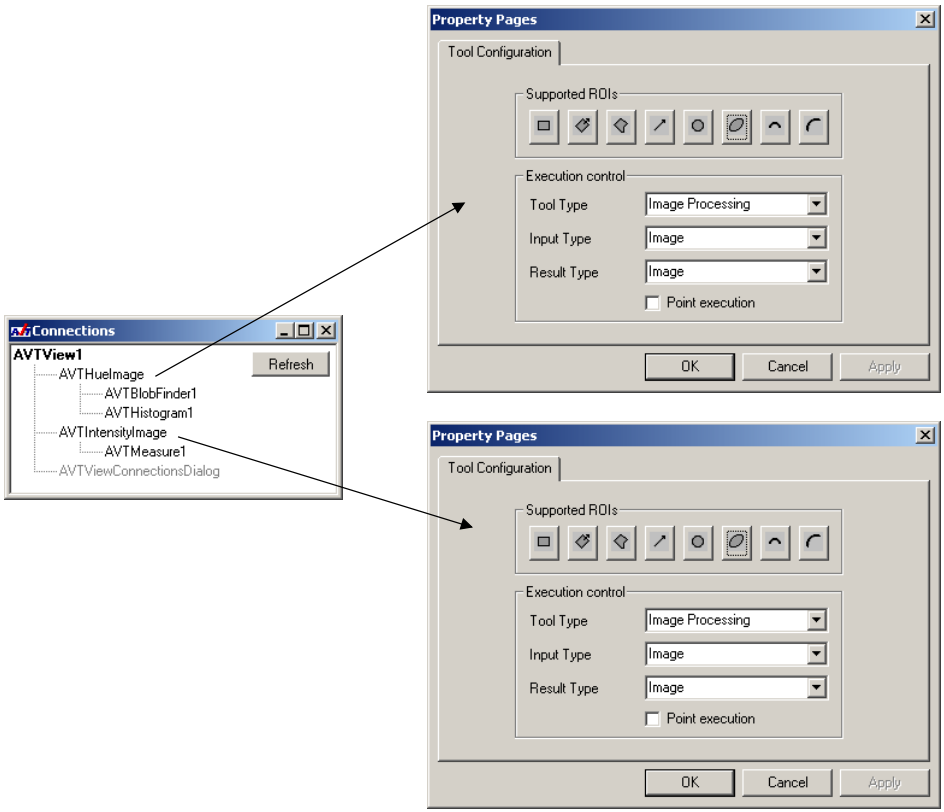


Figure 4.3: Connection and configuration of the two instances of `AVTGenericInterface` in the example.

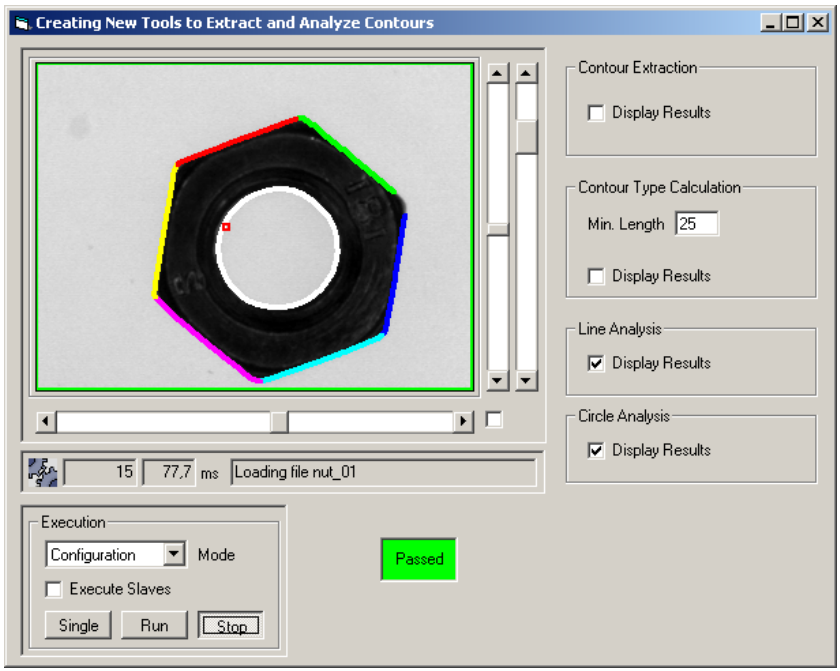


Figure 4.4: Example application extracting and analyzing contour segments.

4.4.2 A Chain of Tools to Extract and Analyze Contours

The Visual Basic example project `contours\advanced_contours.vbp` creates four new ActivVisionTools in charge of extracting and analyzing contours. Figure 4.4 shows the tools in action; the task is to inspect the contours of nuts in the image sequence `misc\nuts1.seq`. Note that the elements to the right of `AVTView` are not really part of the tools but had to be created explicitly using standard Visual Basic elements; as already mentioned, instances of `AVTGenericInterface` are invisible at run time and have no GUI of their own.

The example project is already fully configured; note, however, that the image processing is realized based on HALCON, thus **you need a license for the (virtual) tool ActivHALCON-Foundation to run this project** (or a HALCON license).

!

The task of extracting and analyzing contours is split over the four new tools as follows: Head of the chain is a *vision tool* that extracts contours and outputs them as its iconic results. It is followed by a second *vision tool*, which splits the contours into segments, narrows the segments down to full circles and straight lines and passes them on together with an object feature indicating whether the object is a circle of line segment and two ROI features indicating the number of circles and lines, respectively. Finally, two *result processing tools* follow that work in parallel:

One selects the straight line segments from its input data and calculates their start and end point. The second one selects the circles and calculates their center and radius. Figure 4.5 visualizes this chain.

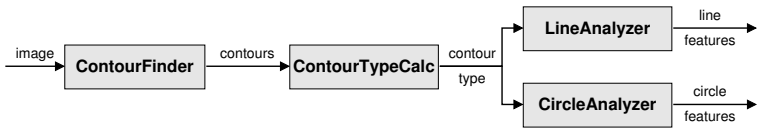


Figure 4.5: The chain of tools extracting and analyzing contours.

In the following, we present the four new tools in more detail.

4.4.2.1 Extracting Contours

Figure 4.6 shows the configuration of the first tool in the chain, `AVTContourFinder`: It is the only one for which you can create ROIs; in these ROIs, it extracts contours and passes them on in form of *XLDs* (eXtended Line Descriptions) as its iconic results.

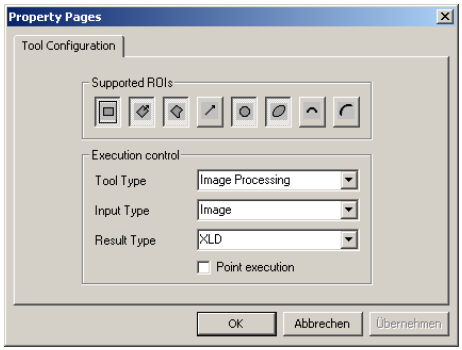


Figure 4.6: The configuration of `AVTContourFinder` .

Because `AVTContourFinder` does not calculate features, its whole activity can be found in a handler for the event `Execute`, which we will discuss step by step below. To use HALCON operators, a global variable of the class `HALCONXLib.HOperatorSetX` is created first:

```
Private hOpSet As New HOperatorSetX
```

As explained in section 4.3.3 on page 113, *vision tools* perform their processing typically within a loop over all ROIs. The handler for `AVTContourFinder`'s event `Execute` is structured accordingly:

```
Private Sub AVTContourFinder_Execute()

  Dim atROI As AVTROIClass

  On Error GoTo ErrorHandler

  If AVTContourFinder.HalconImage Is Nothing Then
    Exit Sub
  End If

  For Each atROI In AVTContourFinder.ROIs
    ' processing, output, display
  Next

  Exit Sub
ErrorHandler:
  Debug.Print Err.Description
End Sub
```

For the processing, the following variables are declared inside the handler:

```
Dim hImageROI As HUntypedObjectX
Dim hBlobRegions As HUntypedObjectX, hBoundaries As HUntypedObjectX
Dim hContourROI As HUntypedObjectX, hImageContourROI As HUntypedObjectX
Dim hImageAmp As HUntypedObjectX, hImageDir As HUntypedObjectX
Dim hEdgePixels As HUntypedObjectX, hClosedEdges As HUntypedObjectX
Dim hContours As HUntypedObjectX
```

In each ROI, contours are extracted using a two-step procedure: First, the image part corresponding to the ROI is accessed. In this image part, blobs are extracted using a fast global thresholding operator. The region around the boundaries of these blobs then serves as ROI for the edge extraction:

```
Set hImageROI = AVTContourFinder.GetReducedImage(atROI, _
                                                 AVTView1.AlignROIs)
Call hOpSet.FastThreshold(hImageROI, hBlobRegions, 0, 128, 10)
Call hOpSet.Boundary(hBlobRegions, hBoundaries, "inner")
Call hOpSet.DilationCircle(hBoundaries, hContourROI, 4.5)
Call hOpSet.ReduceDomain(hImageROI, hContourROI, hImageContourROI)
Call hOpSet.EdgesImage(hImageContourROI, hImageAmp, hImageDir, _
                       "canny", 0.8, "nms", 30, 60)
Call hOpSet.Threshold(hImageAmp, hEdgePixels, 10, 255)
```

The edges are thenclosed and converted into XLD contours:

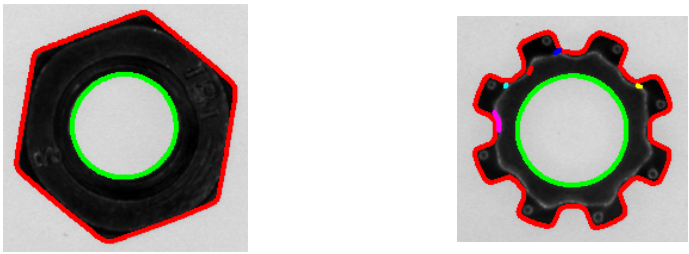


Figure 4.7: Contour segments extracted by `AVTContourFinder` for two example objects.

```
Call hOpSet.CloseEdgesLength(hEdgePixels, hImageAmp, _
                             hClosedEdges, 10, 10)
Call hOpSet.GenContoursSkeletonXld(hClosedEdges, hContours, _
                                   1, "filter")
```

To pass on the extracted segments as output, the method `SetIconicResult` is called as follows:

```
Call AVTContourFinder.SetIconicResult(atROI, hContours)
```

The extracted contours are displayed using the method `AddOverlay`, but only if a corresponding check box is checked (see figure 4.4 on page 121):

```
If CheckCFDisplayResults.Value = 1 Then
  Call AVTContourFinder.AddOverlay(hContours, eDisplayItemGoodObjectsGI)
End If
```

To assure a prompt reaction when you check or uncheck the box, the method `NeedsReexecution` is called within a handler for its event `Click`:

```
Private Sub CheckCFDisplayResults_Click()
  Call AVTContourFinder.NeedsReexecution
End Sub
```

Figure 4.7 depicts the results of `AVTContourFinder` for two objects in the image sequence.

4.4.2.2 Determining the Type of the Contour Segments

The task of the next tool in the chain, `AVTContourTypeCalc`, is to partition the contours received from `AVTContourFinder` into segments and to determine their type. From the segments, only those are selected that correspond to full circles or straight lines with a minimum length, i.e., to those kinds of segments that form the contour of a nut (compare figure 4.7).

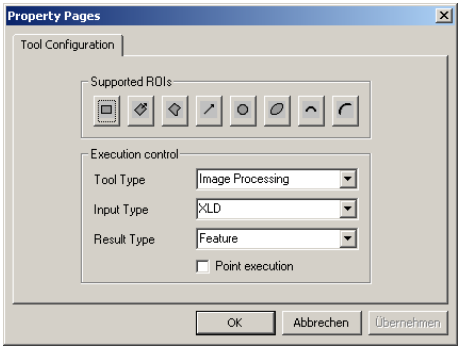


Figure 4.8: The configuration of `AVTContourTypeCalc` .

Figure 4.8 shows the configuration of `AVTContourTypeCalc`. What cannot be seen in the configuration is that, besides features, `AVTContourTypeCalc` also outputs iconic results, which contain the selected contour segments, i.e., the full circles and the long line segments.

In a handler for the event `InitializeFeatures`, three features are declared: Two ROI features indicate, how many lines and (full) circles, respectively, the ROI contains. Furthermore, an object feature indicates the contour type for those contour segments that are passed on as iconic results. The handles of the features are stored in global variables for later use:

```
Private handleNumLines As Integer
Private handleNumCircles As Integer
Private handleType As Integer

Private Sub AVTContourTypeCalc_InitializeFeatures()
  Call AVTContourTypeCalc.SetFeatureNum(3)
  handleNumLines = AVTContourTypeCalc.AddFeature("Number of Lines", _
                                                 eFeatROI, eTypeInt, _
                                                 0, 1000, eUnitDimNone, _
                                                 eEvalIntNumber, 1, 1)
  handleNumCircles = AVTContourTypeCalc.AddFeature("Number of Circles", _
                                                   eFeatROI, eTypeInt, _
                                                   0, 1000, eUnitDimNone, _
                                                   eEvalIntNumber, 1, 1)
  handleType = AVTContourTypeCalc.AddFeature("Contour Type", _
                                             eFeatObject, eTypeWChar, _
                                             "Line", "Circle", eUnitDimNone, _
                                             eEvalIntString, 1, 20)
End Sub
```

In the following, we explain step by step how `AVTContourTypeCalc` partitions the contours

into segments and determines the type in the handler for the event `Execute`. First, some global variables are declared:

```
Private Const cPI As Double = 3.14159265358979
Private Const csContourTypeLine As String = "Line"
Private Const csContourTypeCircle As String = "Circle"
Private iMinLength As Integer
```

Similarly to `AVTContourFinder`, the central element of the event handler is a loop over all ROIs; but as `AVTContourTypeCalc` possesses no own ROIs, it uses those of its predecessor, i.e., those of `AVTContourFinder`:

```
Private Sub AVTContourTypeCalc_Execute()

  Dim atROI As AVTROIClass

  On Error GoTo ErrorHandler

  For Each atROI In AVTContourTypeCalc.GetPredecessorResultsIconicROIs
    ' processing, output, display
  Next

  Exit Sub
ErrorHandler:
  Debug.Print Err.Description
End Sub
```

The processing uses the following variables:

```
Dim hCircles As HUntypedObjectX, hLines As HUntypedObjectX
Dim hTmp As HUntypedObjectX
Dim hInputContours As HUntypedObjectX, hContourSegments As HUntypedObjectX
Dim hLongSegments As HUntypedObjectX, hSegment As HUntypedObjectX
Dim vNumSegments As Variant, i As Integer
Dim vContourType As Variant
Dim vCenterRow As Variant, vCenterCol As Variant, vRadius As Variant
Dim vStartAngle As Variant, vEndAngle As Variant, vPOrder As Variant
Dim iNumLines As Integer, iNumCircles As Integer
Dim hResultingSegments As HUntypedObjectX
Dim sContourTypes() As String
```

For each ROI, the variables storing the extracted circle and lines are reset at the beginning; then, the contours extracted by the predecessor `AVTContourFinder` are accessed:

```
Call hOpSet.GenEmptyObj(hCircles)
Call hOpSet.GenEmptyObj(hLines)
iNumLines = 0
iNumCircles = 0
Set hInputContours = _
  AVTContourTypeCalc.GetPredecessorResultsIconicData(atROI)
```

The contours are split into straight and circular segments; those segments that are shorter than a specified threshold are discarded; the threshold can be modified interactively as shown below.

```
Call hOpSet.SegmentContoursXld(hInputContours, hContourSegments, _
                               "lines_circles", 5, 40, 5)
Call hOpSet.SelectContoursXld(hContourSegments, hLongSegments, _
                              "contour_length", iMinLength, 99999, -1, -1)
```

Now, we loop through the remaining segments to determine their type:

```
Call hOpSet.CountObj(hLongSegments, vNumSegments)
For i = 1 To vNumSegments
  Call hOpSet.SelectObj(hLongSegments, hSegment, i)
  ' determine type
Next
```

The contour segments already indicate their type in form of an attribute. If the attribute describes a segment as a line, it can immediately be added to the list of line segments:

```
Call hOpSet.GetContourGlobalAttribXld(hSegment, "cont_approx", _
                                      vContourType)
If vContourType = -1 Then
  Call hOpSet.ConcatObj(hLines, hSegment, hTmp)
  Call hOpSet.CopyObj(hTmp, hLines, 1, -1)
  iNumLines = iNumLines + 1
```

In contrast, segments described as circles require a closer inspection. For this, we first query the parameters of the segment, i.e., its center point, radius, and the start end end angle:

```
Else
  Call hOpSet.FitCircleContourXld(hSegment, _
                                  "algebraic", -1, 2, 0, 3, 2, _
                                  vCenterRow, vCenterCol, vRadius, _
                                  vStartAngle, vEndAngle, vPOrder)
```

The following code first checks whether the circle is closed; if yes, the segment is added to the list of circles. If the segment corresponds to an open circle, but has a large radius, it can also be approximated by a straight line, therefore we add it to the list of line segments. All other circular segments are discarded.

```
If (vStartAngle = 0#) And (vEndAngle - 2 * cPI < 0.0001) Then
  Call hOpSet.ConcatObj(hCircles, hSegment, hTmp)
  Call hOpSet.CopyObj(hTmp, hCircles, 1, -1)
  iNumCircles = iNumCircles + 1
Else
  If vRadius > 200 Then
    Call hOpSet.ConcatObj(hLines, hSegment, hTmp)
    Call hOpSet.CopyObj(hTmp, hLines, 1, -1)
    iNumLines = iNumLines + 1
  End If
End If
```

At the end of the procesing, the lines and circles are concatenated and then output as the iconic results via the method `SetIconicResult`:

```
Call hOpSet.ConcatObj(hLines, hCircles, hResultingSegments)
Call AVTContourTypeCalc.SetIconicResult(atROI, hResultingSegments)
```

As the next step, the features are prepared for output: First, the number of objects is specified using the method `SetFeatureResultsObjectNum`:

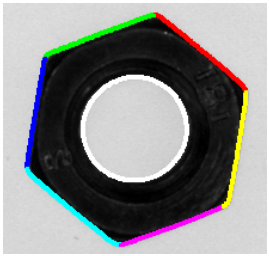
```
vNumSegments = iNumLines + iNumCircles
Call AVTContourTypeCalc.SetFeatureResultsObjectNum(atROI, vNumSegments)
```

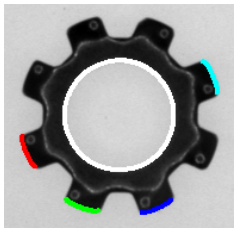
Then, an array variable is filled with strings describing the contour type (line or circle) and used in the call to `SetFeatureResultsObject`:

```
If vNumSegments > 0 Then
  ReDim sContourTypes(vNumSegments - 1)
  For i = 0 To iNumLines - 1
    sContourTypes(i) = csContourTypeLine
  Next
  For i = iNumLines To vNumSegments - 1
    sContourTypes(i) = csContourTypeCircle
  Next
  Call AVTContourTypeCalc.SetFeatureResultsObject(atROI, handleType, _
                                                  sContourTypes)
End If
```

Finally, the two ROI features indicating the number of extracted lines and circles, respectively, are set with the method `SetFeatureResultROI`:

```
Call AVTContourTypeCalc.SetFeatureResultROI(atROI, handleNumLines, _
                                            iNumLines)
Call AVTContourTypeCalc.SetFeatureResultROI(atROI, handleNumCircles, _
                                            iNumCircles)
```





Tool ROI Object	Name	Value
AVTContourTypeCalc	Objects Tool	7
	Good Objects Tool	7
	Bad Objects Tool	0
	Good ROIs	1
	Bad ROIs	0
Rectangle1_1	Objects ROI	7
	Good Objects ROI	7
	Bad Objects ROI	0
	Number of Lines	6
	Number of Circles	1
Object 0	Contour Type	Line
Object 1	Contour Type	Line
Object 2	Contour Type	Line
Object 3	Contour Type	Line
Object 4	Contour Type	Line
Object 5	Contour Type	Line
Object 6	Contour Type	Circle

Tool ROI Object	Name	Value
AVTContourTypeCalc	Objects Tool	5
	Good Objects Tool	5
	Bad Objects Tool	0
	Good ROIs	1
	Bad ROIs	0
Rectangle1_1	Objects ROI	5
	Good Objects ROI	5
	Bad Objects ROI	0
	Number of Lines	4
	Number of Circles	1
Object 0	Contour Type	Line
Object 1	Contour Type	Line
Object 2	Contour Type	Line
Object 3	Contour Type	Line
Object 4	Contour Type	Circle

Figure 4.9: Results of `AVTContourTypeCalc` for two example objects.

Figure 4.9 shows the results of `AVTContourTypeCalc` for two objects in the image sequence: For the nut, 6 line segments and a circle are found as desired, while most of the contour segments of the ring are discarded because them being either too short or open circles.

The extracted lines and circles are displayed in `AVTView` using the method `AddOverlay`, with different color groups for the two types to allow to distinguish them. As with `AVTContourFinder`, the display can be enabled or disabled individually via a corresponding check box:

```
If CheckCTCDisplayResults.Value = 1 Then
  Call AVTContourTypeCalc.AddOverlay(hLines, eDisplayItemGoodObjectsGI)
  Call AVTContourTypeCalc.AddOverlay(hCircles, eDisplayItemGoodObjectsGII)
End If
```

Furthermore, because `AVTContourTypeCalc` produces features, it also supports the display of error results and the highlighting of selected objects by calling the method `SetFeatureObjects` with the extracted segments as parameter:

```
Call AVTContourTypeCalc.SetFeatureObjects(atROI, hResultingSegments)
```

As can be seen in figure 4.4 on page 121, two standard Visual Basic elements allow to configure

AVTContourTypeCalc interactively. The influence of the check box governing the display of results was already described; what remains to say is that to assure a prompt reaction to your checking or unchecking the box, the method NeedsReexecution is called within a handler for its event Click:

```
Private Sub CheckCTCDisplayResults_Click()
  Call AVTContourTypeCalc.NeedsReexecution
End Sub
```

With the text box, you can set the minimum length the contour segments must have in order to be selected. In the following handler to its event Change, the current value of the box is stored in a global variable, if it is valid; then, NeedsReexecution is called to force an update:

```
Private Sub TextCTCMinLength_Change()

  If TextCTCMinLength.Text = "" Or Not IsInteger(TextCTCMinLength.Text) Then
    iMinLength = 26
  Else
    iMinLength = CInt(TextCTCMinLength.Text)
    If iMinLength < 1 Then
      iMinLength = 1
    End If
  End If
  Call AVTContourTypeCalc.NeedsReexecution

End Sub
```

The following auxiliary method checks whether the text box contains an integer value:

```
Private Function IsInteger(vText As Variant) As Boolean
  Dim iVal As Integer
  On Error GoTo ErrorCase

  IsInteger = True
  iVal = vText
  Exit Function

ErrorCase:
  IsInteger = False
End Function
```

4.4.2.3 Analyzing Straight Line Segments

As the result of AVTContourTypeCalc, the contour segments have been narrowed down to long line segments and full circles and their number and type has been made available in form

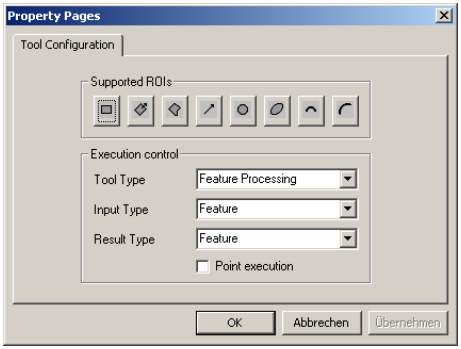


Figure 4.10: The configuration of `AVTLineAnalyzer` and `AVTCircleAnalyzer` .

of features. The two *result processing tools* described in the following can now start to analyze the segments more closely.

Figure 4.10 shows the configuration of the two tools. As we already noted for `AVTContourTypeCalc`, the configuration does not tell the whole truth: While `AVTContourTypeCalc` outputs not only features as stated in the configuration, but also iconic results in form of the extracted segments, `AVTLineAnalyzer` and `AVTCircleAnalyzer` do not only require features as its input, but also the just mentioned iconic results of `AVTContourTypeCalc` in order to analyze them. This is no problem, though, because all tools receive the iconic data as input, as described in section 3.1 on page 54. In fact, the two *result processing tools* continue in the tradition of their predecessor and output not only features but also iconic results: `AVTLineAnalyzer` outputs the analyzed line segments, `AVTCircleAnalyzer` the analyzed circles. Thus, it would be possible to continue the chain with further tools that need both features and iconic results, e.g., a tool that groups parallel lines into pairs to calculate suitable grasping points from a robot gripper.

The task of `AVTLineAnalyzer` is to extract the line segments from the iconic results provided by `AVTContourTypeCalc` and to calculate features describing them. In a handler for the event `InitializeFeatures`, therefore, four object features are declared, describing the coordinates of the start and end point of the line segment. The handles of the features are stored in global variables for later use (error handling omitted):

```
Private handleStartRow As Integer
Private handleStartCol As Integer
Private handleEndRow As Integer
Private handleEndCol As Integer
```

```
Private Sub AVTLineAnalyzer_InitializeFeatures()

  Call AVTLineAnalyzer.SetFeatureNum(4)
  handleStartRow = AVTLineAnalyzer.AddFeature("Start Row", _
                                              eFeatObject, eTypeDouble, _
                                              0, 240, eUnitDimMeter, _
                                              eEvalIntNumber, 1, 1)
  handleStartCol = AVTLineAnalyzer.AddFeature("Start Column", _
                                              eFeatObject, eTypeDouble, _
                                              0, 360, eUnitDimMeter, _
                                              eEvalIntNumber, 1, 1)
  handleEndRow = AVTLineAnalyzer.AddFeature("End Row", _
                                            eFeatObject, eTypeDouble, _
                                            0, 240, eUnitDimMeter, _
                                            eEvalIntNumber, 1, 1)
  handleEndCol = AVTLineAnalyzer.AddFeature("End Column", _
                                            eFeatObject, eTypeDouble, _
                                            0, 360, eUnitDimMeter, _
                                            eEvalIntNumber, 1, 1)

End Sub
```

Because `AVTLineAnalyzer` not only processes features but also iconic results, the central element of the handler for its event `Execute` is again a loop over the ROIs for which iconic results exist:

```
Private Sub AVTLineAnalyzer_Execute()

  Dim atROI As AVTROIClass

  For Each atROI In AVTLineAnalyzer.GetPredecessorResultsIconicROIs
    ' processing, output, display
  Next

End Sub
```

Nevertheless, the following code at the beginning checks whether the also required feature results exist:

```
Dim atToolResult As AVTToolResult
Dim bResultsFound As Boolean

If AVTLineAnalyzer.GetPredecessorResultsFeatureData Is Nothing Then
 Exit Sub
End If

For Each atToolResult In AVTLineAnalyzer.GetPredecessorResultsFeatureData
 If atToolResult.Name = "AVTContourTypeCalc" Then
  bResultsFound = True
  Exit For
 End If
Next
If bResultsFound = False Then
 Exit Sub
End If
```

The processing uses the following variables:

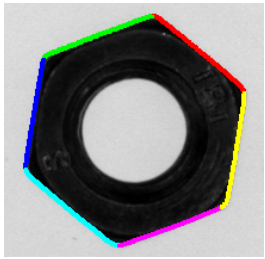
```
Dim iNumLines As Integer
Dim hInputSegments As HUntypedObjectX, hLines As HUntypedObjectX
Dim vStartRows As Variant, vStartCols As Variant
Dim vEndRows As Variant, vEndCols As Variant
Dim vNr As Variant, vNc As Variant, vDist As Variant
```

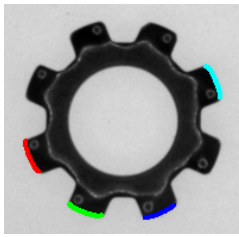
For each ROI, the variable storing its line segments is reset at the beginning.

```
Call hOpSet.GenEmptyObj(hLines)
```

Then, the lines extracted by the predecessor `AVTContourTypeCalc` are accessed. This task could be solved by looping over all segments and checking the feature describing the type for each one. Instead, the following code exploits the fact that lines come first in the iconic results; therefore, it first queries the overall number of lines via the corresponding ROI feature and then simply copies the desired segments from the iconic results:

```
iNumLines = atToolResult.ROIValue(atROI.Name, handleNumLines)
If iNumLines > 0 Then
  Set hInputSegments = _
    AVTLineAnalyzer.GetPredecessorResultsIconicData(atROI)
  Call hOpSet.CopyObj(hInputSegments, hLines, 1, iNumLines)
End If
```





Object Id	Start Row	Start Column	End Row	End Column
0	223.767	452.792	160.016	375.613
1	161.022	373.641	195.712	281.366
2	198.013	281.951	290.849	266.661
3	293.066	265.643	357.265	343.873
4	355.717	346.364	323.359	432.639
5	322.972	436.082	226.219	452.196

Object Id	Start Row	Start Column	End Row	End Column
0	291.711	185.206	317.437	196.500
1	345.743	223.158	355.804	250.142
2	357.022	287.733	345.959	313.187
3	253.186	352.905	227.208	340.910

Figure 4.11: Results of `AVTLineAnalyzer` for two example objects.

Now, all line segments are analyzed simultaneously:

```
Call hOpSet.FitLineContourXld(hLines, "huber", -1, 0, 5, 2, _
                              vStartRows, vStartCols, vEndRows, vEndCols, _
                              vNr, vNc, vDist)
```

As already noted, `AVTLineAnalyzer` passes on the line segments as its iconic results:

```
Call AVTLineAnalyzer.SetIconicResult(atROI, hLines)
```

As all lines have been analyzed simultaneously, the calculated feature values are automatically stored in form of arrays; thus, they can be set with a single call of `SetFeatureResultsObject`. Before setting the output features, the number of objects is specified:

```
Call AVTLineAnalyzer.SetFeatureResultsObjectNum(atROI, iNumLines)
Call AVTLineAnalyzer.SetFeatureResultsObject(atROI, handleStartRow, _
                                             vStartRows)
Call AVTLineAnalyzer.SetFeatureResultsObject(atROI, handleStartCol, _
                                             vStartCols)
Call AVTLineAnalyzer.SetFeatureResultsObject(atROI, handleEndRow, _
                                             vEndRows)
Call AVTLineAnalyzer.SetFeatureResultsObject(atROI, handleEndCol, _
                                             vEndCols)
```

Figure 4.11 shows the results of `AVTLineAnalyzer` for two objects in the image sequence.

The display of results is realized similarly to `AVTContourTypeCalc`: The lines are displayed in `AVTView` using the method `AddOverlay`, with the display being enabled and disabled via a

corresponding check box. Besides, the method SetFeatureObjects is called to support the display of error results and the highlighting of selected objects:

```
If CheckLADisplayResults.Value = 1 Then
  Call AVTLineAnalyzer.AddOverlay(hLines, eDisplayItemGoodObjectsGI)
End If
Call AVTLineAnalyzer.SetFeatureObjects(atROI, hLines)
```

To assure a prompt reaction when you check or uncheck the box governing the display of results, the method NeedsReexecution is called within a handler for its event Click:

```
Private Sub CheckLADisplayResults_Click()
  Call AVTLineAnalyzer.NeedsReexecution
End Sub
```

4.4.2.4 Analyzing Circles

The second *result processing tool*, AVTCircleAnalyzer, is constructed along the same lines as the first one, AVTLineAnalyzer; therefore, we describe it in less detail.

The task of AVTCircleAnalyzer is to extract the circles from the iconic results provided by AVTContourTypeCalc and to calculate three features describing them: the coordinates of the center point and the radius. The features are declared in a handler for the event InitializeFeatures as follows (global variable declaration and error handling omitted):

```
Private Sub AVTCircleAnalyzer_InitializeFeatures()

  Call AVTCircleAnalyzer.SetFeatureNum(3)
  handleCenterRow = AVTCircleAnalyzer.AddFeature("Center Row", _
                                                 eFeatObject, eTypeDouble, _
                                                 0, 240, eUnitDimMeter, _
                                                 eEvalIntNumber, 1, 1)
  handleCenterCol = AVTCircleAnalyzer.AddFeature("Center Column", _
                                                 eFeatObject, eTypeDouble, _
                                                 0, 360, eUnitDimMeter, _
                                                 eEvalIntNumber, 1, 1)
  handleRadius = AVTCircleAnalyzer.AddFeature("Radius", _
                                              eFeatObject, eTypeDouble, _
                                              0, 240, eUnitDimMeter, _
                                              eEvalIntNumber, 1, 1)
End Sub
```

Analogously to AVTLineAnalyzer, the handler for AVTCircleAnalyzer's event Execute revolves around a loop over the ROIs for which iconic input results exist, but also checks whether the input includes the required feature results. For the processing, the following variables are

declared:

```
Dim iNumCircles As Integer, iNumLines As Integer
Dim hInputSegments As HUntypedObjectX, hCircles As HUntypedObjectX
Dim vCenterRows As Variant, vCenterCols As Variant, vRadiuses As Variant
Dim vStartAngles As Variant, vEndAngles As Variant, vPOrders As Variant
```

For each ROI, the variable storing its circles is first reset and then filled with the circles extracted by the predecessor AVTContourTypeCalc, exploiting the fact that AVTContourTypeCalc passes on the extracted segments sorted by type, i.e., first the lines, then the circles. Thus, AVTCircleAnalyzer queries the number of extracted lines and circles via the corresponding ROI features and then copies the desired segments from the iconic results. Finally, all circles are analyzed simultaneously:

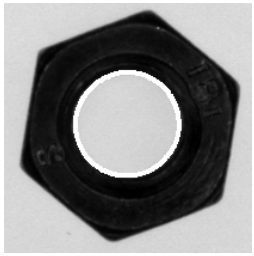
```
Call hOpSet.GenEmptyObj(hCircles)
iNumCircles = atToolResult.ROIValue(atROI.Name, handleNumCircles)
If iNumCircles > 0 Then
  Set hInputSegments = _
    AVTCircleAnalyzer.GetPredecessorResultsIconicData(atROI)
  iNumLines = atToolResult.ROIValue(atROI.Name, handleNumLines)
  Call hOpSet.CopyObj(hInputSegments, hCircles, _
                      iNumLines + 1, iNumCircles)
End If
Call hOpSet.FitCircleContourXld(hCircles, "ahuber", -1, 2, 0, 3, 2, _
                                vCenterRows, vCenterCols, vRadiuses, _
                                vStartAngles, vEndAngles, vPOrders)
```

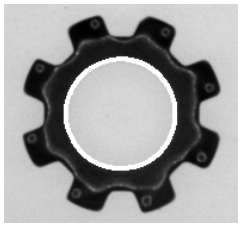
AVTCircleAnalyzer's consists of iconic results (the circle segments) and the calculated object features:

```
Call AVTCircleAnalyzer.SetIconicResult(atROI, hCircles)
Call AVTCircleAnalyzer.SetFeatureResultsObjectNum(atROI, iNumCircles)
Call AVTCircleAnalyzer.SetFeatureResultsObject(atROI, handleCenterRow, _
                                               vCenterRows)
Call AVTCircleAnalyzer.SetFeatureResultsObject(atROI, handleCenterCol, _
                                               vCenterCols)
Call AVTCircleAnalyzer.SetFeatureResultsObject(atROI, handleRadius, _
                                               vRadiuses)
```

Figure 4.12 shows the results of AVTCircleAnalyzer for two objects in the image sequence. The display of results is realized as described for AVTLineAnalyzer.

Based on the feature results of AVTLineAnalyzer and AVTCircleAnalyzer, the example project inspects the objects in the image sequence misc\nuts1.seq by specifying conditions for ActivDecision: Only those objects are evaluated as "okay" whose contour is "nut-shaped", i.e., consists of exactly six line segments and one full circle. Furthermore, the size of the nut is





Object Id	Center Row	Center Column	Radius
0	257.459	357.698	43.466

Object Id	Center Row	Center Column	Radius
0	273.551	268.805	46.052

Figure 4.12: Results of `AVTCircleAnalyzer` for two example objects.

checked by evaluating the radius of the circle. Step through the image sequence to see the effect.

4.4.3 Using Calibration and Alignment Information

In this section we show how to use calibration and alignment information in an ActivGenericInterface. Both types of information are part of the input data of all tools (see also section 3.1 on page 54). `AVTGenericInterface` provides additional properties and methods that facilitate the access and the use of this information.

In the following, we present two Visual Basic example projects. In both projects, a new *inspection tool* is created, which receives the current position of the mouse within the image as input data, similarly to ActivZoom. The task of the new tool is to display this position in `AVTView`, next to the mouse pointer. In the first example, the position is also converted into the currently selected unit, based on the calibration information. In the second example, it is also transformed into the coordinates of the reference image. Please refer to the User's Manual for ActivGeoCalib and the User's Manual for ActivAlignment for more information about calibration and alignment in general.

Both projects are already fully configured. You need no ActivHALCONFoundation or HALCON license to run them.

4.4.3.1 Example 1: Converting Data Into World Coordinates

The Visual Basic example project `calibration\advanced_calibration.vbp` is an extension of an example described in the User's Manual for ActivGeoCalib. It already contains 3D calibration information that allows to measure in a plane that is oriented at an angle with respect to the camera; note that **you need a license for ActivGeoCalib to run the example**. In the

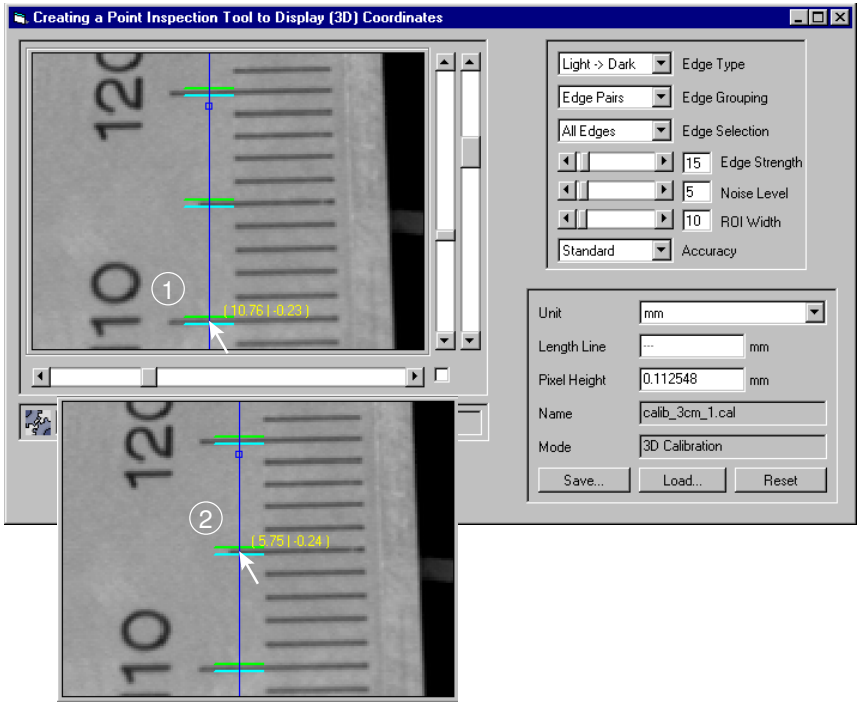


Figure 4.13: The new *inspection tool* displays the world coordinates of the mouse next to it in `AVTView` at two positions.

image `calib\rule_01`, a calliper rule is placed in the plane of measurement, and the distances between its markings are measured using ActivMeasure (see figure 4.13).

As described above, the example contains a new ActivVisionTool named `AVTShowCoordinates` whose task is to display the current position of the mouse in the currently selected unit in `AVTView`. Figure 4.13 depicts the result with the mouse being pointed at two neighboring markings. As you can see, the column coordinate is almost constant while the row coordinate differs by 5mm as desired. Just start the project and experiment with it, e.g., create a 2D calibration using `AVTViewCalibration` and observe the effect; the file containing the 3D calibration can be found in the subdirectory `images\calib`. In the following, we explain the code behind this new tool; figure 4.14 shows its configuration.

Because `AVTShowCoordinates` is a *point tool*, its functionality is placed into a handler for the event `ExecutePoint`; as parameters, this event passes the pixel coordinates of the current mouse position in the image:

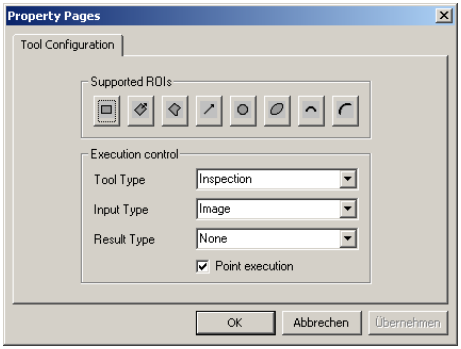


Figure 4.14: The configuration of the new tool `AVTShowCoordinates` .

```
Private Sub AVTShowCoordinates_ExecutePoint(iRow As Integer, iCol As Integer)

  ' processing and display

End Sub
```

The first step in this handler is to access the calibration data. `AVTGenericInterface`'s property `Calibration` returns them in form of an `ActivVTools.AVTCalibClass`:

```
Dim atCalibData As AVTCalibClass

Set atCalibData = AVTShowCoordinates.Calibration
```

Now, we want to convert the position of the mouse from pixels into world coordinates. For this, the class `ActivVTools.AVTCalibClass` provides the method `ImagePointTo3DUnit` (see the **Reference Manual** for more information about this method and other useful methods of `ActivVTools.AVTCalibClass`). Please note that the method expects parameters of the type `Variant`, to allow the simultaneous conversion of multiple points. While you can use other types for the input parameters, **you must use** `Variant` **for the output parameters**, even if you convert but a single point, otherwise the output will be `0`! Thus, `AVTShowCoordinates` converts the mouse position with the following code:

```
Dim vXCoord As Variant, vYCoord As Variant
Dim dRowCoord As Double, dColCoord As Double

Call atCalibData.ImagePointToWorldUnit(iRow, iCol, vXCoord, vYCoord)
dRowCoord = vYCoord(0)
dColCoord = vXCoord(0)
```

As you can see, it is very simple to convert data into world coordinates. In fact, much more code

is devoted to displaying the data at a suitable position in AVTView. First, a string is assembled containing the coordinates:

```
Dim sCoordinates As String

sCoordinates = "( " & Format$(dRowCoord, "Fixed") & " | " & _
               Format$(dColCoord, "Fixed") & " )"
```

This string is now to be displayed next to the mouse pointer in AVTView. To display text, AVTGenericInterface provides the method AddOverlayText. Besides the text, you can specify the display position in the image (in pixel coordinates) and some optional parameters, e.g., the desired alignment of the text or whether the text is to be displayed immediately (see also the **Reference Manual**). In our example application, however, it is not easy to find suitable parameter values: If the mouse is pointed at the upper left corner, e.g., the text should be displayed below and to the right of the mouse pointer to be fully visible, while the contrary is true if the mouse is pointed at the lower right corner.

In the example, this problem is solved by using a different display position and alignment for each of the four quarters of the display window: In the upper left quarter, the text is displayed left-aligned, below and to the right of the mouse pointer, in the lower right quarter it is displayed right-aligned, above and to the left of the mouse pointer and so forth:

```
Dim iPosRow As Integer, iPosCol As Integer

If InUpperHalfOfWindow(iRow) = True Then
  iPosRow = iRow + 10# / AVTView1.ScaleY
Else
  iPosRow = iRow - 25# / AVTView1.ScaleY
End If
If InLeftHalfOfWindow(iCol) = True Then
  iPosCol = iCol + 25# / AVTView1.ScaleX
  Call AVTShowCoordinates.AddOverlayText(sCoordinates, iPosRow, iPosCol, _
                                         eTextLeft, True)
Else
  iPosCol = iCol - 10# / AVTView1.ScaleX
  Call AVTShowCoordinates.AddOverlayText(sCoordinates, iPosRow, iPosCol, _
                                         eTextRight, True)
End If
```

As you can see, the display position is calculated by adding or subtracting an offset to the position of the mouse pointer. These offsets are divided by the the current zooming factor(s), which you can modify interactively using the right one of the two sliders to the right of AVTView (see figure 4.13 on page 138).

The code for the two functions InUpperHalfOfWindow and InLeftHalfOfWindow can be found in the project. These tasks are not easy to solve because positions are given in image

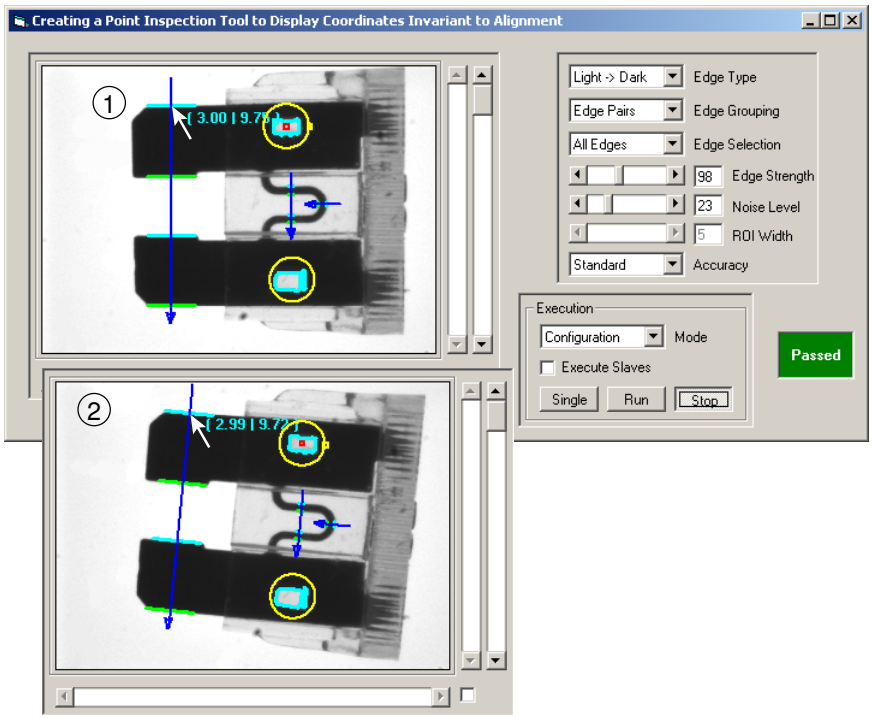


Figure 4.15: `AVTShowCoordinates` now displays the (world) coordinates of the mouse relative to the coordinate system of the reference image.

coordinates, while the division into quarters relates to the image part that is currently displayed in `AVTView`, i.e., the part you choose with the three sliders to the right and below `AVTView`.

4.4.3.2 Example 2: Using Alignment Information

In the Visual Basic example project `alignment\advanced_alignment.vbp`, the new tool `AVTShowCoordinates` presented above is extended: Besides the calibration, it also takes into account alignment information. The project itself is an extension of an example described in the User's Manual for ActivDecision; in it, ActivAlignment is used to position three measurement ROIs over a car fuse (see figure 4.15). Note that **you need licenses for ActivAlignment, ActivMeasure, and ActivDecision to run the example**

!

In an ActivVisionTool, alignment information is typically used in two ways: The obvious one is to align ROIs to a moving part; here, data specified in the reference image is transformed into the current image, following the movement of the alignment anchor. The other one is to transform

features calculated in the current image back into the reference image; you enable this mode by checking the box ☑ `Make Features Invariant` in `AVTAlignmentUse`.

When creating a new tool using ActivGenericInterface, you need not worry about the first way of alignment: When you call `AVTGenericInterface`'s method `GetReducedImage` to access the image within an ROI, existing alignment information is automatically applied to the ROI, if the alignment of ROIs is enabled in general and if you don't disable the alignment "locally" via a corresponding (optional) parameter of the method. Please note that the data structures representing the ROIs themselves still contain the original data, i.e., the position and orientation of the ROI in the reference image.

In the example project, we show how to apply the second way of alignment. The task is to display the current position of the mouse relative to the coordinate system of the reference image. Figure 4.15 on page 141 shows the result: The mouse is pointed at the same part of the fuse; even if the fuse moves, the displayed position stays the same. Just start the project and experiment with it, e.g., uncheck ☑ `Make Features Invariant` in `AVTAlignmentUse` and observe the effect. In the following, we explain the code behind this functionality. As in the previous example, it can be found in the handler for the event `ExecutePoint`:

```
Private Sub AVTShowCoordinates_ExecutePoint(iRow As Integer, iCol As Integer)
  ...
End Sub
```

You can access the alignment information via `AVTGenericInterface`'s property `AffineTransformation`, where it is stored in form of an `ActivVTools.AVTAffineClass`:

```
Dim atAlignmentData As AVTAffineClass

Set atAlignmentData = AVTShowCoordinates.AffineTransformation
```

To tranform data back into the reference image, `ActivVTools.AVTAffineClass` provides the method `InvTransPointDouble` (see the **Reference Manual** for more information about this method and other useful methods of `ActivVTools.AVTAffineClass`). This method has an optional parameter by which you can disable the transformation. At first glance, this parameter seems to be unnecessary, because why should one call the method and then forbid the transformation. The advantage of this parameter is however, that if you disable the transformation, the output parameters are set to the input parameters; thus, you can use the output parameters in the following code independent of whether the transformation takes place or not.

`AVTShowCoordinates` employs this mechanism by performing the transformation only if the alignment is enabled generally and if the box ☑ `Make Features Invariant` is checked:

```
Dim dRefRow As Double, dRefCol As Double

Call atAlignmentData.InvTransPointDouble(iRow, iCol, dRefRow, dRefCol, _
                                         (AVTView1.AlignROIs And _
                                          AVTView1.MakeFeatureInvariant))
```

Then, the tool continues as described in the previous example, i.e., with converting the position into world coordinates:

```
Call atCalibData.ImagePointToWorldUnit(dRefRow, dRefCol, vXCoord, vYCoord)
```

The display is realized as described in the previous example.

Chapter 5

Advanced Image Acquisition

This chapter discusses advanced aspects of image acquisition. We expect the reader to have read the User's Manual for ActivView.

5.1 Parallel Grabbing 146
5.2 Grabbing from Multiple Cameras 147
5.2.1 Port Switching 147
5.2.2 Grabbing from Multiple Cameras Simultaneously 150

5.1 Parallel Grabbing

As described in the User's Manual for ActivView, section 1.3.3 on page 12, by default an ActivVisionTools application parallelizes the grabbing and the actual processing of an image. This means that while the application processes an image, the image acquisition device is already grabbing the next one; thus, when the application finishes processing, the next image is typically already available and waiting to be processed. This mechanism is very important if the application is to work at frame-rate, i.e., if you must assure that every image is processed in time, and if this rate of images is determined externally, i.e., by the process.

However, there are also cases in which sequential grabbing is more suited, i.e., in which the next image should be grabbed after the application has finished processing the current one. In this cases it's the ActivVisionTools application which determines when images are to be grabbed, not an external process. A typical example of such a case is the button `Single` in `AVTViewFG`: If you click on the button, the image is to be grabbed at the moment of the click; you would be very surprised if you got the image grabbed while the previous image was being processed, which might be some time ago!

You can enable and disable the parallel grabbing mode by setting `AVTView`'s property `FGParallelGrabbing` to `'True'` and `'False'`, respectively. You can do this interactively in `AVTViewProperties`, which can be opened by clicking on `AVTView` with the right mouse button and selecting `Properties` in the appearing context menu (see the User's Manual for ActivView, section 2.3 on page 20 for more information about this tool). The property's value is stored in the application's DSC file along with other parameters of `AVTView`; if the DSC file contains no entry for this property, the default value, i.e., `'True'`, is used.

As we already saw, even if parallel grabbing is enabled it is not automatically used for all grabbing actions. To be exact, if enabled, parallel grabbing is

automatically used

- when you execute ActivVisionTools continuously, independent of whether the continuous execution was started interactively, e.g., by clicking the button `Run` in `AVTViewExecute`, or via the programming interface as described in section 3.2 on page 59.

possibly used

- when you trigger a single execution cycle via the programming interface by calling `AVTView`'s method `ExecuteAllToolsOnce`. As described in section 3.2 on page 59, in order to allow parallel grabbing you must specify `'TRUE'`for the second parameter of this method.
- when you call `AVTView`'s method `GrabImage` (see section 3.2 on page 59).

not used

- when you trigger a single execution cycle interactively, i.e., via the button `Single` in

AVTViewFG or the corresponding ones in AVTViewExecute or ActivGeoCalib.

- when images are loaded from files.

As already noted, in the parallel grabbing mode after each grab a new grab is started automatically. In some application, however, you may want to start the new grab explicitly, e.g., to synchronize multiple cameras. This can be achieved by calling AVTView's method GrabImageStart.

5.2 Grabbing from Multiple Cameras

With many of the image acquisition devices supported by ActivVisionTools, grabbing from multiple cameras can be realized with multiple instances of AVTView as described in section 3.6.1.1 on page 92: By simply connecting each of the instances to one of the cameras using AVTViewFG. For some acquisition devices, however, this simple interactive method does not work; instead, you must insert some lines of code. The following sections describe example projects that can serve as a template for such cases.

5.2.1 Port Switching

Some image acquisition devices access multiple cameras via the so-called *port switching*. Here, you open the connection to the device only once and then dynamically switch between ports and the cameras connected to them. This means that you cannot grab from multiple cameras simultaneously but only from one after the other. Typically, such acquisition devices only have one A/D converter.

You can switch between ports interactively using AVTViewFGParameter as described in the User's Manual for ActivView, section A.5.2 on page 58 or via the programming interface as described in the same manualin section A.5.4 on page 62. In this section, we explain how to distribute the images fom different cameras to different instances of AVTView. The necessary code is contained in the project multiple_cam_portswitching\advanced_multiple_cam_portswitching.vbp, which allows to set the port for master and slave via text boxes, to grab from both cameras via an instance of AVTViewExecuteSimple configured as a single-step button, or to grab only from a single camera via a standard button (see figure 5.1).

The main problem with port switching and multiple AVTViews is that only one AVTView (the master) actually connects to the acquisition device and grabs images. Thus, if the master grabs an image from the other AVTView's camera, it must send it to the other AVTView and restore its own image. The slave is executed in the mode 'Use Locally Stored Image' (also see section 3.6.1.3 on page 96), so that it used the image that has been set by the master. This means

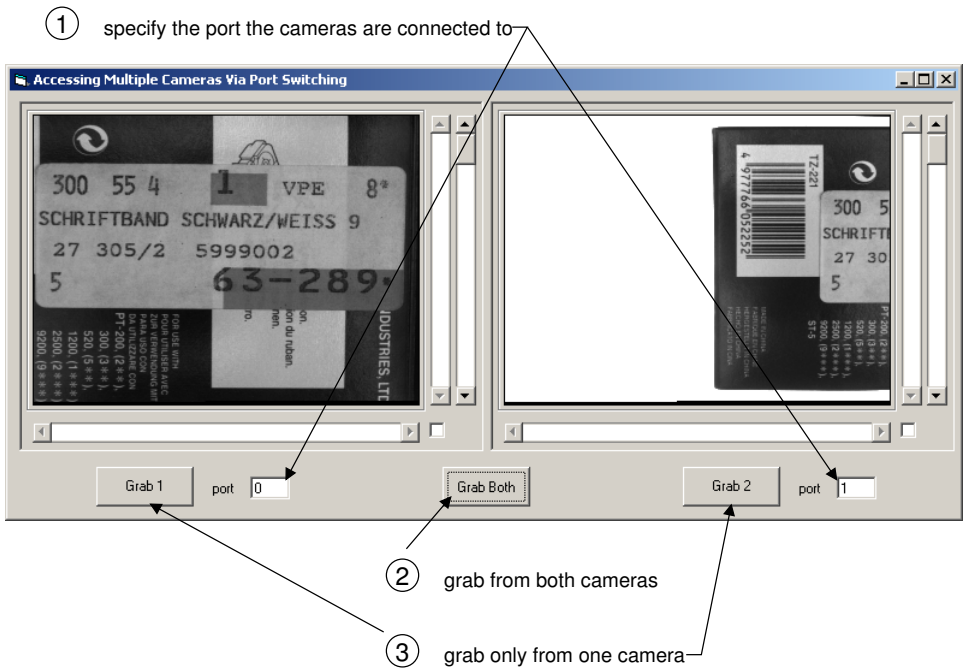


Figure 5.1: Using the example project for port switching.

that the slave does not grab images itself, thus, you need to set up the image acquisition only for the master.

First of all, global variables are defined to store the state of the application: whether the master is grabbing for the slave and whether images are to be acquired from both or only one camera:

```
Private bGrabbingForSlave As Boolean
Private bGrabbingFromBoth As Boolean
```

In the event handler for `ImageGrabbed`, a (static) variable is used to save the current master image:

```
Private Sub AVTView1_ImageGrabbed(hImage As HALCONXLib.IHUntypedObjectX, _
                                  ByVal lError As Long)

Static hCurrentMasterImage As HUntypedObjectX
```

If the image that has just been acquired is for the master, it is saved in the static variable:

```
  If bGrabbingForSlave = False Then
    Set hCurrentMasterImage = hImage
```

If images are to be acquired from both cameras, the port is switched via a method (that is described below), then the master is called to acquire an image, using parallel grabbing (see section 5.1 on page 146) if the application is running continuously:

```
    If bGrabbingFromBoth = True Then
      bGrabbingForSlave = True
      Call SetPort(iSlavePort)
      If AVTView1.RunState = False Then
        Call AVTView1.GrabImage(False)
      Else
        Call AVTView1.GrabImage(True)
      End If
    End If
```

After the image has been acquired for the slave, the event handler gets active again. Now, the corresponding global variable signals that the image is for the slave. The image is therefore passed on, and the master image is restored to the saved one. Finally, the port is switched back and the state variable is reset:

```
  Else  ' bGrabbingForSlave = True
    Set AVTView2.HalconImage = hImage
    Set AVTView1.HalconImage = hCurrentMasterImage
    Call SetPort(iMasterPort)
    bGrabbingForSlave = False
  End If

End Sub
```

The actual port switching is encapsulated in a method to facilitate the error handling:

```
Private Sub SetPort(iPort As Integer)

On Error GoTo ErrorHandler
  Call AVTView1.SetFGParameterValue("port", iPort)
  Exit Sub

ErrorHandler:
  Call MsgBox("Please use a frame grabber that supports port switching!", _
              vbExclamation, "Application Error")
End Sub
```

If only an image for the master is to be acquired, the slave is “switched off” by setting its slave mode to `'Independent'` (see section 3.6.1.4 on page 98) before starting an execution cycle:

```
Private Sub CommandGrab1_Click()

  Call SetPort(iMasterPort)
  bGrabbingFromBoth = False
  AVTView2.SlaveMode = eSlaveIndependent
  Call AVTView1.ExecuteAllToolsOnce
  AVTView2.SlaveMode = eSlaveProcessCurrentOnExecution
  bGrabbingFromBoth = True

End Sub
```

If only an image for the slave is to be acquired, the slave is also switched onto the mode `'Independent'`. After calling the master to grab an image, the slave is then executed alone:

```
Private Sub CommandGrab2_Click()

  Call SetPort(iSlavePort)
  AVTView2.SlaveMode = eSlaveIndependent
  bGrabbingForSlave = True
  bGrabbingFromBoth = False
  Call AVTView1.GrabImage(False)
  Call AVTView2.ExecuteAllToolsOnce(False)
  AVTView2.SlaveMode = eSlaveProcessCurrentOnExecution
  bGrabbingFromBoth = True
End Sub
```

5.2.2 Grabbing from Multiple Cameras Simultaneously

Some image acquisition devices, e.g., `Leutron` or `INSPECTA`, allow to grab images simultaneously from multiple cameras that are connected to different ports of a single device, which has multiple A/D converters. If you trigger the grabbing in this mode, you will not immediately get the desired behavior: In `AVTView`, an image is displayed that looks like a combination of the individual images, but only one of the images is processed. The reason for both effects is that the ActivVisionTools application does not receive multiple images, but a so-called *multi-channel image*, i.e., an image containing multiple sub-images.

We already encountered this type of images in section 3.4.2 on page 73 and section 3.6.1.3 on page 96: Both RGB and HSI color images are represented by 3-channel images. The methods presented there to access, process, and display different channels of an RGB image can of course also be used to access, process, and display the individual images stemming from the different cameras. In the following, we show how to distribute images that are simultaneously grabbed from two cameras on two instances of `AVTView`.

`multiple_cam_simultaneous\advanced_multiple_cam_simultaneous.vbp` contains

the necessary code. In comparison to the port switching described above, it is very simple: In the handler for the event `ImageGrabbed` of the master instance of `AVTView`, the two images are extracted from the multi-channel image and then passed on to master and slave. The slave is executed in the mode `'Use Locally Stored Image'` (also see section 3.6.1.3 on page 96), so that it used the image that has been set in the event handler. This means that the slave does not grab images itself, thus, you need to set up the image acquisition only for the master.

```
Private Sub AVTView1_ImageGrabbed(hImage As HALCONXLib.IHUntypedObjectX, _
                                  ByVal lError As Long)

  Dim Image1 As HUntypedObjectX
  Dim Image2 As HUntypedObjectX

  If AVTView1.HalconImageChannelCount = 2 Then
    Set Image1 = AVTView1.HalconImageChannel(1)
    Set Image2 = AVTView1.HalconImageChannel(2)
  Else
    Set Image1 = hImage
    Set Image2 = hImage
  End If

  Set AVTView1.HalconImage = Image1
  Set AVTView2.HalconImage = Image2

End Sub
```

For information about how to configure the connection to an image acquisition device to grab images simultaneously from multiple cameras, please refer to the online documentation of the HALCON image acquisition interfaces, which can be found under `http://www.halcon.com/image-acquisition`. Parameters listed in the section *Parameters for open_framegrabber* are set in `AVTViewFG`, those listed in the section *Parameters for set_framegrabber_param* in `AVTViewFGParameter`. Note that ActivVisionTools and the HALCON image acquisition interfaces use a slightly different terminology.